Benjamin Cole

Implementing a Java to WebAssembly Compiler

Computer Science Tripos – Part II

Clare College

May 2020

Declaration of Originality

I, Benjamin Cole of Clare College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

I, Benjamin Cole of Clare College, am content for my dissertation to be made available to the students and staff of the University.

Signed: [illegible]

Date: 07/05/2020

Proforma

Candidate Number:	**2390D**
Project Title:	**Implementing a Java to WebAssembly Compiler**
Examination:	**Computer Science Tripos – Part II, June 2020**
Word Count:	**11996**[1]
Lines of Code:	**7790**[2]
Project Originator:	Dr Timothy M. Jones
Supervisor:	Dr Timothy M. Jones

Original Aims of the Project

The original aim of the project was to produce a compiler from a subset of Java to WebAssembly and validate that its generated code is correct. Core objectives were to support basic language constructions such as expressions, statements and functions, and to support dynamic memory allocation. Extensions were to support inheritance and dynamic polymorphism, and to include garbage collection.

Work Completed

I implemented the compiler, using ANTLR as a parser and supporting object-oriented programming features including classes, arrays, inheritance, polymorphism and generic types. I wrote WebAssembly subroutines for memory allocation and expansion and for garbage collection. I wrote tests for and benchmarked the generated code. The success criteria were met.

Special Difficulties

None.

[1]Computed by Overleaf's word count tool for chapters 1–5.

[2]Computed by `cloc src scripts sample_programs tests`, excluding whitespace and comments. See `https://github.com/AlDanial/cloc`.

Contents

List of Figures

Chapter 1

Introduction

WebAssembly is a new binary instruction format for a stack-based virtual machine [21]. The language was created by the World Wide Web Consortium to serve as an efficient, portable format for distributing performance-integral parts of an application, while remaining inter-operable with existing JavaScript applications. It is now an official web standard and is the fourth language to be supported natively in web browsers, alongside HTML, CSS and JavaScript [22]. This dissertation presents *JavAssembler*, a compiler from a subset of Java to WebAssembly, written in Java.

1.1 Motivation

The main motivation for the development of WebAssembly was to improve the performance of web applications. Until recently, JavaScript was the only possible language for distributing client-side code. For multiple reasons, JavaScript can be quite a slow language to execute.

Modern JavaScript runtimes, such as Google's V8 engine used in the Node.js runtime and the Google Chrome web browser [20], execute JavaScript by translating it into bytecode, then interpreting the bytecode until the Just-In-Time (JIT) compiler deems a particular function worth compiling to native code. As a result, the application has to be delivered in source-code form, and the runtime environment has to spend time translating the application from JavaScript into bytecode. While WebAssembly does still need parsing and translating, the language is far closer to that of an underlying machine, dramatically reducing the amount of work required and hence offering a significant performance improvement.

Another hindrance to JavaScript's performance is its dynamic type-system, which limits the amount of speculative optimisation that can be performed. Variables can be freely declared and assigned values without type information, and a variable holding a value of one type may later be reassigned to one of a different type. In a statically-typed language like C++ or Java, if we know that the variables `x` and `y` are integers, then evaluating the expression `x + y` is guaranteed to complete without throwing an exception or causing any side effects. The lack of typing information in JavaScript means that JavaScript runtimes are offered no such guarantees; instead, a JavaScript interpreter would

```java
int f(int a, int b) {
    return a * b + 10;
}
```

(a) Java source code

```wasm
(func $f (param $a i32) (param $b i32)
  (result i32)
  local.get $a
  local.get $b
  i32.mul
  i32.const 10
  i32.add
)
```

(b) Equivalent WebAssembly code

Figure 1.1: An example function written in both Java and WebAssembly.

have to perform several checks before actually computing the addition [1]. WebAssembly is a strongly-typed language, with only four possible types: 32- and 64-bit integers and floats. The type of every variable must be declared explicitly and will never change. This enables JIT compilers like that used in the V8 engine to map WebAssembly code to native machine code without having to insert additional type-safety logic.

WebAssembly is not a replacement for JavaScript; rather, it serves as a sandboxed execution environment fully contained within a JavaScript runtime. Instead of writing all client-side code in JavaScript, developers can write performance-intensive subroutines in a compiled high-level language like C++ or Rust, which they then compile into WebAssembly modules that their existing JavaScript code can interface with directly.

As of February 2020, the TIOBE Index ranks Java as the world's most popular programming language, and the language continues to grow in popularity [19]. The development of an efficient compiler from Java to WebAssembly would enable millions of developers to use their existing skills to write more efficient web applications.

WebAssembly was designed primarily for languages that would otherwise compile to native assembly code, while Java was designed to compile to bytecode for the Java Virtual Machine (JVM). The JVM is object-aware and provides applications with dynamic memory allocation and memory management; in contrast, WebAssembly is neither object-aware nor does it provide any garbage collection. Translating from a garbage-collected language to a non-garbage-collected runtime poses many interesting problems.

1.2 Overview of WebAssembly

WebAssembly sits at a level of abstraction considerably lower than that of Java. There are no expressions: computation is performed by pushing values to and popping them from the operand stack. There are neither objects nor arrays: the language simply provides developers with an expandable region of linear memory. This gives compiler writers lots of freedom to implement higher-level language features in their own way. Figure 1.1 shows an example program written in both languages.

While WebAssembly is lower-level than most conventional programming languages, it is still higher-level than most actual machine instruction sets, like x86 and ARM, and requires a software runtime for execution. The call stack resembles that of a high-level language, using named variables rather than memory offsets. Call frames are managed automatically: the compiler does not need to worry about calling conventions. Function calls and jumps can be performed symbolically rather than relying on memory addresses. These differences make writing a compiler to WebAssembly uniquely different to writing a compiler to a more conventional target language.

1.3 Existing Work

WebAssembly is far from the first attempt at improving the performance of browser-executed code. Rather than creating a new language, `asm.js` is designed to be an easily optimisable, low-level subset of JavaScript that can be targeted by compilers of higher-level languages like C [5]. `asm.js` avoids some of the performance penalty of JavaScript's dynamic type-system by using type coercions. However, a 2019 study found that C programs compiled to WebAssembly perform on average 34% faster in Google Chrome than the same programs compiled to `asm.js` [27].

There are also existing solutions for compiling Java code into WebAssembly. For example, one could use the Polyglot compiler [18] combined with the JLang back-end [14] to emit LLVM IR code, then use LLVM's WebAssembly back-end to emit WebAssembly code. Alternatively, one could use the JWebAssembly compiler to translate Java Bytecode to WebAssembly [16]. However, by writing the entire pipeline myself, I was able to provide a complete one-stage solution, and to learn more about compilers in the process.

1.4 Project Summary

In this project, I implemented a compiler from a subset of Java to WebAssembly, covering language features like inheritance and dynamic polymorphism and supporting dynamic memory allocation and garbage collection. Throughout the project I explored the design decisions required and the performance trade-offs involved.

Chapter 2

Preparation

Before starting any software project, it is essential to consider its requirements carefully and plan how the program will be implemented. In this chapter, I begin by analysing the requirements for the compiler and explain how these can be satisfied. I then discuss some of the difficulties of targeting WebAssembly, and how I planned to address them. Finally, I describe the software engineering techniques and workflow I used to ensure that the code I produced was of the highest possible quality.

2.1 Requirements Analysis

In my project proposal, I identified some key success criteria of the project. Here I discuss how these can be tested and verified.

- **(Core)** *Support compiling simple single-file programs consisting of stack variables and static methods*

  This is the minimum viable product, covering only a tiny subset of Java. However, achieving this actually required the implementation of a significant proportion of the compiler. To compile even basic programs, JavAssembler needed a parser, an AST builder and a code-generation stage. I decided I could verify that this requirement had been satisfied by producing tests for the relevant language features to check that the semantics of Java had been implemented correctly.

- **(Core)** *Support dynamic memory allocation for heap objects*

  Being able to create and pass around objects is essential in modern programming. Restricting programmers to exclusively stack-allocated variables would prevent them from using many common data structures like lists and trees, making it needlessly difficult to implement many algorithms. This is why I chose to include dynamic memory allocation as a core requirement.

  To satisfy this requirement I needed to implement a memory allocator and update the grammar and code generation stages to support the relevant language constructions, such as the `new` keyword. I decided I would confirm that dynamic memory allocation was working correctly by writing a linked-list implementation and demonstrating that new nodes could be allocated and appended at runtime.

- **(Extension)** *Support object-oriented programming features such as inheritance and dynamic polymorphism*

  Java is widely known for its object-oriented paradigm. To capture the language properly, JavAssembler needed to support classes and standard OOP features like inheritance and dynamic polymorphism. Inheritance is useful because developers can take existing code and extend it, adding in the features they need while reusing the existing code for everything else. Dynamic polymorphism is useful because it enables developers to create different objects with their own behaviours, but all accessible under a common interface.

  Supporting inheritance would involve significant additions to the type system. I needed to implement a *class inheritance hierarchy graph* and a *topological sort* algorithm so that classes' internal layouts could be generated correctly. Supporting dynamic polymorphism required building a virtual table for each class. I decided that I could test this by writing code that exposes dynamically polymorphic behaviour, then writing tests to validate that the correct method is called.

- **(Extension)** *Include garbage collection in the generated code*

  If a program keeps allocating more memory for new objects without releasing that used by inaccessible older objects, it will quickly run out of memory. C++ addresses this by forcing the programmer to delete objects explicitly; Java instead uses a garbage collector to scan the stack and heap to delete objects automatically once they have fallen out of scope and are no longer accessible. I wanted to include garbage collection in JavAssembler because working with objects is an essential part of object-oriented programming.

  As well as needing to write a garbage-collection algorithm, I also needed to modify the in-memory representation of objects so that the garbage collector was able to access the stack. I chose to use *Cheney's algorithm* [26], a copying collection algorithm that repeatedly copies objects between two heaps. A significant difficulty is that WebAssembly's main call stack cannot be inspected by the current function, so I was forced to store heap references in a second manually-managed stack that resides in WebAssembly's linear memory space.

  I decided I could test the garbage collector by writing a program that repeatedly allocates and discards objects—far more than could fit into the heap without garbage collection—and executing it to check that memory is being recycled.

2.2 Compiler Design

Most compilers operate using a pipeline of distinct stages. Each stage transforms the program to remove some form of abstraction. Pipeline stages broadly fall into three layers:

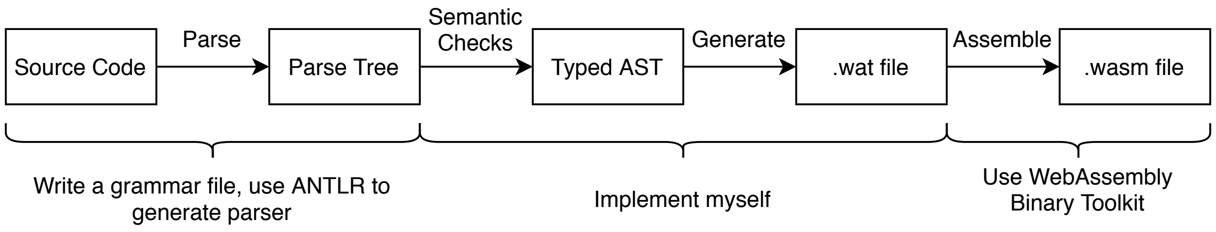


Figure 2.1: Overview of JavAssembler's pipeline.

- The *front-end* is the entry point to the compiler. This is the stage responsible for reading in the source code and parsing it to produce an in-memory representation of the program.
- The *intermediate stage* is where the program is analysed for correctness, and where higher-level constructs are replaced with compositions of lower-level data types and operations. Some compilers also perform optimisations in their intermediate stage.
- The *back-end* is the final stage of the compiler. This is where the in-memory representation of the program is translated into the target assembly language.

Modern state-of-the-art compilers have astonishingly many stages. The advantage of having a deep pipeline is that the transformations at each stage are minimal and easy to understand. Modular code is also easier to maintain: you can change one intermediate representation without having to rewrite the rest of the compiler. However, given the time frame available, I decided to keep the pipeline short, giving me more time to work on supporting high-level language features. The pipeline I chose to use is shown in Figure 2.1.

Another design decision was the garbage-collection algorithm to use. Countless types of garbage-collection algorithm exist, the simplest being *reference counting*. Reference counting works by keeping count of the number of inward references to each object, deleting objects when this falls to zero. The main disadvantage is that reference counting fails to delete inaccessible cycles.

Tracing algorithms work by traversing the *reachability graph*. In the reachability graph, nodes correspond to heap objects and the presence of an edge indicates that one object holds a reference to another. An object is *reachable* if there is a path to it starting from the call stack; objects that are unreachable are safe for deletion. *Copying collection* is a type of tracing algorithm in which the heap is partitioned into two spaces and the set of accessible objects copied between them whenever garbage collection is required, effectively deleting any objects that the garbage collector could not find. I chose to implement *Cheney's algorithm* [26], a copying collection algorithm that uses $\mathcal{O}(1)$ additional space, because this avoids the possibility of a stack overflow.

2.3 Starting Point

The goal of the project was to implement the whole pipeline of the compiler. However, I did make use of existing libraries for certain features. Rather than implementing a parser myself, I used ANTLR, a parser-generation library [3]. This saved time and reduced the number of tests I had to write. For similar reasons, I also used the Apache Commons CLI library [4] for parsing command-line arguments.

Rather than emitting pure binary code in the form of a `.wasm` file, I chose to emit human-readable code in the *WebAssembly textual representation*, a `.wat` file, which made the generated code easier to read and debug. I then used the WebAssembly Binary Toolkit [23], imported as a JavaScript library, to assemble the generated code into a true WebAssembly binary, ensuring that this happened before any benchmarks to avoid impacting the measured performance.

2.4 Work to be Completed

With the requirements specified and the starting point identified, the next step was to determine the main high-level software components needed.

Parser	I needed to implement the compiler front-end so that it could produce a *parse-tree* representation of the program. To do this, I needed to write a grammar describing the syntax of Java and use ANTLR to generate a parser for it.
AST Data Structures	I needed to produce data structures to represent the *abstract syntax tree* of a program, along with suitable interfaces. This required writing a class for each construction in Java's abstract syntax.
AST Builder	The goal of the AST builder is to convert parse trees into abstract syntax trees. This required writing a class that traverses the parse tree and converts it into an AST.
Type System	A *type system* was required so that Java expressions and variables could be mapped to appropriate WebAssembly types, and so that ill-typed programs could be rejected at compile-time. I needed to implement a scheme for computing typing judgements for all expressions.
Function Table	The function table is needed at compile-time to resolve static method calls using their name and type signature. I needed to implement a data structure that could be used to identify functions from their name and argument types.
Virtual Tables	Virtual tables are needed at runtime for resolving non-static class methods dynamically. I needed to implement a compile-time representation of these tables so that method calls could be resolved to virtual-table indices, as well as a subroutine to write them to the generated WebAssembly module.

Semantic Analysis	The purpose of the semantic-analysis stage is to check that the program is legal. For example, programs that reference undeclared variables should be rejected. I decided that, rather than incorporating this as a separate stage in the compiler pipeline, I would add semantic checks as part of the AST construction process.
Memory Allocator	The generated WebAssembly code needs a way to dynamically allocate memory for created objects. I needed to write a memory allocator to handle this. For simplicity, I decided that I would implement a bump allocator directly as a WebAssembly function that could be bundled with the generated code.
Garbage Collector	To ensure that inaccessible objects are deleted, I needed to write a garbage collector. Like the memory allocator, I decided that it would be easiest to write this directly as a WebAssembly function that could be called by the memory allocator if more free space is required.
Code Generation	The code-generation stage of the compiler is responsible for actually generating the WebAssembly code. I needed to implement an algorithm that could translate each method into WebAssembly and bundle these in a WebAssembly module.

2.5 Problems to Solve

This section discusses some of the WebAssembly-specific problems that I needed to overcome.

2.5.1 WebAssembly Addressing Restrictions

WebAssembly only permits memory lookups to its linear memory space, not to its call stack. As a result, functions have no way to access values stored in call frames below them in the stack. Garbage collection relies on the ability to determine which heap objects are reachable so that inaccessible objects can safely be deleted. This restriction meant that using WebAssembly local variables to store heap references would render tracing garbage collection impossible.

The solution I designed was to split the call stack into two regions. For primitive-typed local variables, JavAssembler would allocate a standard WebAssembly local variable; for non-primitive local variables, it would instead assign call-frame offsets in a second, manually-managed stack, residing at the opposite end of the linear memory space from the heap like in the x86 instruction set. The garbage collector could then use the second stack to determine object reachability, as shown in Figure 2.2. Another advantage is that this scheme avoids the need to disambiguate between values and pointers.

```
public class Demo {
  private int x;
  private double y;
}

public class Example {
  private static void f()
  {
    int a = 12;
    Demo d = new Demo();
  }
}
```

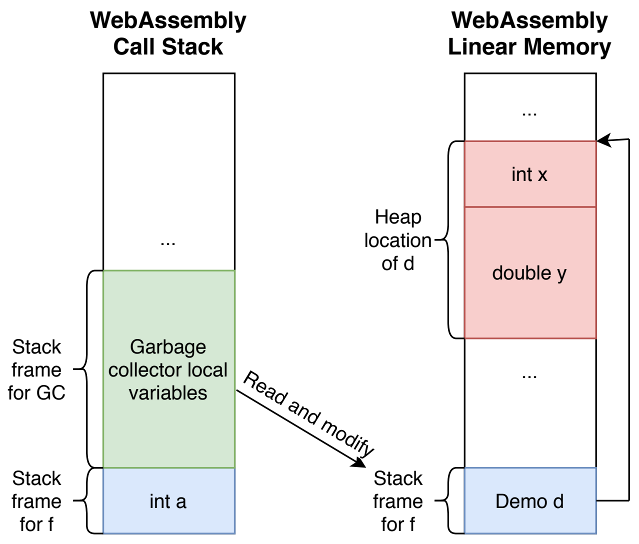


Figure 2.2: An example Java program and its representation in WebAssembly's memory model.

2.5.2 Sandboxing

By design, WebAssembly provides a sandboxed execution environment for computation. WebAssembly programs can never access memory addresses outside the linear memory space assigned to them by their JavaScript container. While this provides convenient security guarantees for web users who may not trust the code they have downloaded, it does increase the complexity of managing multiple `.wasm` files.

Typically, compilers generate an *object file* for each file of source code, then *link* the object files together afterwards using a *symbol table*. The problem is that, without specific JavaScript code to set it up, separate WebAssembly files do not share heap memory. This renders call-by-reference, as used by Java for non-primitive values, impossible between functions in different modules.

Emscripten, an existing WebAssembly compiler backend for the LLVM system [7], handles this by emitting both a `.wasm` file and a JavaScript runtime in which to execute it [8]. The WebAssembly code produced by the compiler is not sufficient on its own and requires the JavaScript caller to set up imports from other files and supply it with memory. However, to make the generated WebAssembly code easier to integrate on its own with an existing application, and to avoid the complexity of having to generate JavaScript code as well as WebAssembly code, I decided to handle compiling programs that consist of multiple source files just by concatenating them into a single WebAssembly file, bundled with the hand-written memory-management subroutines.

2.6 Software Engineering Techniques

Being a large project, adhering to good software engineering techniques was essential. Not only does this make it easier to write the code, it also ensures that the code is maintainable and extensible should anyone wish to add new features in the future.

2.6.1 Code Style

Developers reportedly spend ten times as much time reading code as they do writing it [28]. For this reason, good code should be easily readable by anyone—not just its author. I followed the Android Open Source Project Java Style Guide [2] for formatting my code[1], and included numerous comments and docstrings to explain the purpose of each class and method.

2.6.2 Development Workflow

While developing the project, I used Git for version control. At all times, I kept a copy of the repository both on my laptop and on my Google Drive cloud account. I also pushed every commit to a private repository on GitHub. Keeping backups in multiple places limited the amount of work that could be lost in the event of damage to my laptop or the outage of one of these services.

2.6.3 Testing

The most important property of a compiler is the correctness of the code it generates. To that end, testing was a prominent feature of my development process. I used a test-driven development workflow: before writing a new feature, I first wrote a suite of tests to verify the correctness of the implementation. I used Jest, a JavaScript testing library [13], to write tests for the generated code to ensure that it adheres to Java's semantics. I also wrote unit tests using JUnit [15] and Mockito [17]. There are a total of 157 tests, all of which JavAssembler passes. I also performed benchmarking using Benchmark.js [6], a JavaScript benchmarking library.

2.6.4 Licensing

My code depends on several libraries and frameworks. Details of these libraries' licenses are summarised below.

- **WebAssembly Binary Toolkit** - Apache License 2.0
- **Apache Commons CLI** - Apache License 2.0
- **ANTLR** - BSD License
- **Jest** - MIT License
- **JUnit 5** - Eclipse Public License 2.0
- **Mockito** - MIT License
- **Benchmark.js** - No named license but permission is explicitly granted for anybody to use and distribute

All of these licenses permit me both to use and redistribute them with my project.

[1]Excluding attribute naming rules

2.7 Technologies to Learn

This was my first time writing a compiler, and my first time using WebAssembly. I familiarised myself with the language and surrounding framework by reading the documentation and writing toy programs to test out language features. Building on my knowledge from the *IB Compiler Construction* course, I researched garbage-collection algorithms and considered their suitability for the WebAssembly environment. As discussed previously, I decided to use Cheney's algorithm. This was also my first time using a parser generator. I had to learn to write grammar files, and I read the documentation to discover how to interact with the parse tree that ANTLR generates.

2.8 Computer Setup

For this project, I needed to install tools for both Java and JavaScript development. For Java development, I set up Gradle and installed IntelliJ, my preferred Java IDE. For the JavaScript and WebAssembly components, I installed NodeJS and NPM, and I wrote the code using Visual Studio Code.

Chapter 3

Implementation

With the preparation complete, the next stage was to implement the compiler. This chapter begins with an overview of the compilation process, before examining each step in more detail. It discusses how JavAssembler reads and parses the source code, and how it checks it for correctness; it explains how I implemented complex language features like inheritance and polymorphism, and how JavAssembler handles memory management; finally, it describes how the WebAssembly source code is generated.

3.1 Repository Overview

Following is a summary of the repository layout. I have supplied scripts for compiling JavAssembler and the example programs, and for running the tests and benchmarks, all of which are explained in `README.md`.

```
JavAssembler
├── example_programs ....Contains example code used by the tests and benchmarks
├── scripts
├── src ...............................Contains the source code of JavAssembler
│   ├── antlr ...............................Contains the ANTLR grammar files
│   │   └── parser
│   │       ├── JavaFile.g4
│   │       └── JavaFileTokens.g4
│   ├── java
│   │   ├── ast ....Contains data structures for representing the AST of a subroutine
│   │   ├── codegen .......Contains all code related to writing the WebAssembly file
│   │   ├── errors ........Defines a set of custom exceptions to be thrown internally
│   │   ├── parser ..................Contains classes relating to parsing source files
│   │   └── util ........Includes various utility classes used throughout the compiler
│   └── wasm-lib ............Contains the hand-written WebAssembly subroutines
│       ├── arrays.wat
│       ├── alloc.wat
│       ├── gc.wat
│       └── globals.wat
├── tests ........................................Contains the JavaScript tests
└── README.md
```

3.2 How JavAssembler Works

Compilation in JavAssembler forms a sequence of distinct steps.

1. First, JavAssembler uses the ANTLR-generated parser to parse each source file into a *parse tree*.
2. The `ClassHierarchyBuilder` class reads the class definition in each file and uses this to build a *class inheritance hierarchy graph*.
3. JavAssembler then computes a *topological sort* of this graph to find a serial order in which to process each class.
4. The compiler performs a second pass over the classes, in which all attributes and method signatures are recorded to the *class table* and *function table* respectively. Parse trees of each method are queued for compilation.
5. With the class table and function table complete, the parse tree for each method is converted to an *abstract syntax tree* by the `ASTBuilder` class. This class uses ANTLR's *visitor* pattern to traverse the parse tree of each method.
6. The AST data structures perform semantic analyses on their arguments, such as checking that they are well-typed. Function and method calls are resolved to function-table entries and virtual-table offsets; non-primitive object references are resolved to stack offsets.
7. Finally, the *code-generation* stage translates each AST into WebAssembly code, wrapping the generated functions in a WebAssembly module.

This chapter examines these stages in more detail, highlighting some of the design decisions that were made along the way.

3.3 The Front-End

I began the project by implementing JavAssembler's front-end. This required writing grammar files to describe the structure of a Java program. I separated the grammar description into two files to increase modularity: `JavaFileTokens.g4` for the lexical rules and `JavaFile.g4` for the parser rules. I added ANTLR as a Gradle dependency so that the parser would be automatically generated as part of the compilation process.

3.3.1 The Lexing Stage

The lexing stage is the first part of the parser. ANTLR uses `JavaFileTokens.g4` to produce a *deterministic finite automaton* for converting the stream of characters in the source file into a stream of more meaningful tokens that can be understood by the parser. Figure 3.1 shows a sample of the rules used.

A convenient feature of ANTLR is that tokens can be directed to separate streams called *channels*. A parser can select which channels to listen to and which to ignore. ANTLR provides a special `skip` channel that automatically ignores any tokens it receives. I removed all comments and whitespace in the input by catching them with regular expressions and automatically redirecting them to the `skip` channel, shown in Figure 3.2.

3.3.2 The Parsing Stage

With the source file converted to a sequence of tokens, the next stage is to parse those tokens into a more meaningful tree structure. Like most other programming languages, Java programs can be thought of as being tree-structured. Figure 3.4 shows an example statement in Java, and Figure 3.5 shows how it can be represented as a parse tree. The job of the parsing stage is to build this structure using just the sequence of tokens from the lexer. To implement the parser, I created the file `JavaFile.g4`, which encodes all syntactically legal input files as a *context-free grammar*. Figure 3.3 shows some examples of these rules.

A common difficulty in defining context-free grammars for languages is that grammars can be *ambiguous*—that is, a string may have multiple valid parses. For example, the string `"1+2*3"` could plausibly be parsed as $(1 + 2) \times 3$ or $1 + (2 \times 3)$, which clearly are not equivalent. ANTLR resolves this by treating the production rules as being in priority order. Writing the rule for multiplication and division before the rule for addition and subtraction ensured that expressions are parsed and hence evaluated correctly. I validated

```
EQUALS: '=' ;
PLUS: '+' ;
...
IDENTIFIER: [a-zA-Z_][a-zA-Z_0-9]* ;
```

Figure 3.1: A sample of the lexical rules.

```
COMMENT: '//' ~[\r\n]* -> skip ;
WHITESPACE: [ \t\r\n]+ -> skip ;
```

Figure 3.2: The rules for ignoring comments and whitespace.

```
expr: literal
    | LPAREN expr RPAREN
    | MINUS expr
    | expr multiplicativeBop expr
    | expr additiveBop expr
    | ...
    ;

multiplicativeBop
    : op=MULTIPLY
    | op=DIVIDE
    ;

additiveBop
    : op=PLUS
    | op=MINUS
    ;
```

Figure 3.3: Some of the production rules for expressions.

this by writing a test for this particular expression and checking that it evaluates to 7, not 9.

3.3.3 Constructing the AST

The output of the parser is a tree-like representation of the program, but unfortunately this contains lots of unnecessary additional 'syntactic sugar' with no semantic content. For example, braces help the parser to delineate code blocks but serve no purpose once the tree structure has been built. JavAssembler's next step is to traverse the parse-tree of each method and translate it into an *abstract syntax tree* representation.

To implement this, I created the `ASTNode` interface, along with a data structure for each type of language construction, such as `IfStatement` and `BinaryOperatorExpression`. I created a class called `ASTBuilder` to traverse the parse tree using the visitor pattern, returning an `ASTNode` object for each sub-tree it processes. For modularity, I separated the processing of expressions into another class, `ExpressionVisitor`, which `ASTBuilder` calls into whenever it encounters an `Expression` node. Figure 3.6 shows the AST representation of the program shown in Figure 3.4.

```
if (cond) {
    return x + 1;
} else {
    return f(10);
}
```

Figure 3.4: An example statement in Java.

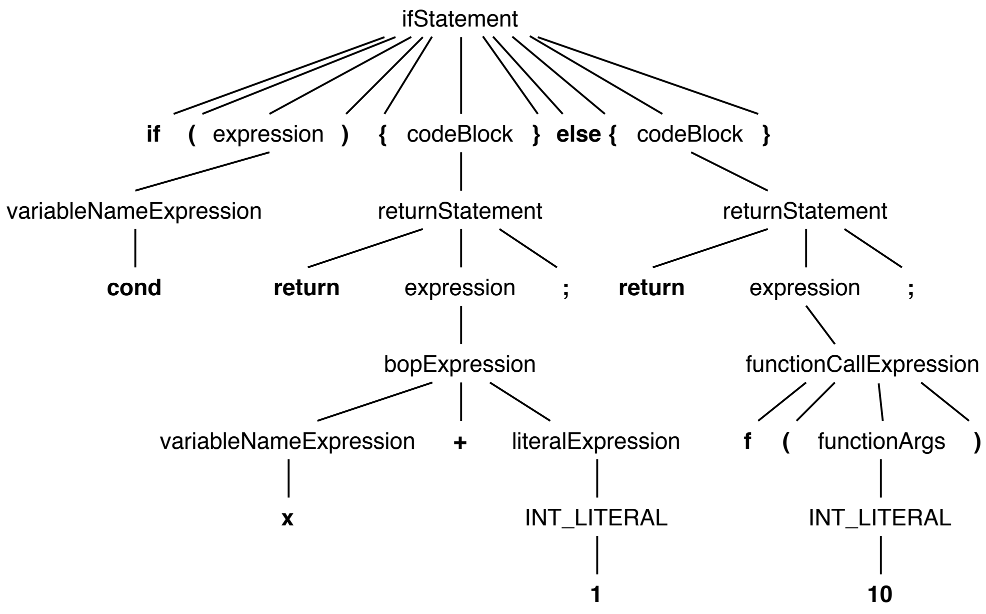


Figure 3.5: A parse-tree representation of the statement in Figure 3.4.

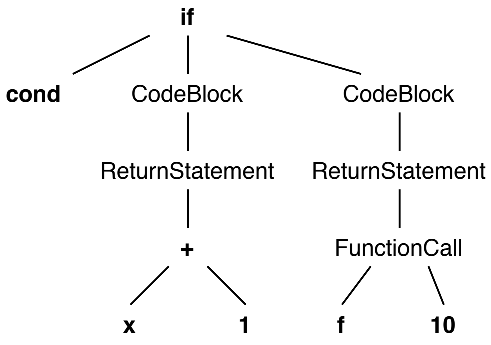


Figure 3.6: An abstract syntax tree representation of the statement in Figure 3.4.

3.4 Semantic Analysis

With the program parsed, the next stage is to check it for correctness. Many compilers treat semantic analysis as a distinct step from parsing, but, to reduce the amount of code required, I instead decided to implement semantic checks as part of the AST construction phase. The semantic-analysis stage performs various checks to ensure that the input program is legal, and will terminate the compilation process with an appropriate error message if not.

3.4.1 Variable Scoping

Scoping is an important concept in many programming languages. Scopes are essentially just sets of local variables spanning some region of the program, such as the body of a method or a while-loop. When a local variable is declared, the variable becomes bound to its containing scope and will not be visible outside it. Scopes can be nested: variables in outer scopes can be accessed from inner scopes, but variables from inner scopes cannot be accessed from outside. An example of scoping is shown in Figure 3.7. An important consequence is that, in this example, the variable `y` should not be accessible at program points 4 and 5, even if the branch of the if-statement is taken.

To capture this behaviour, I implemented a class called `VariableScope`. An outermost scope is created for the parameters to the method; all scopes within this then have a `parent` attribute pointing to the scope in which they are contained. The `ASTBuilder` class maintains these scopes in a stack while it traverses the parse tree. Each time it encounters a construction that creates a new scope, such as an if-statement or a while-loop, it pushes a new `VariableScope` object to the stack, and pops it again once it has finished visiting that sub-tree. If the AST builder encounters a variable declaration, it registers it with the scope on the top of the stack. The `VariableScope` object records the type of that variable and the underlying WebAssembly local variable or stack offset to which it will later be mapped.

Using this structure allows identifiers to be looked up recursively. This is used for code-generation—by which point variables will have been allocated to WebAssembly local

```
void example(int N, boolean cond) {
  doSomethingElse();          [1]
  int x = 1;                  [2]
  if (cond) {
    float y = 2.0f;           [3]
  }
  for (int i=0; i<N; i++)     [4]
  {
    double z = 3.0;           [5]
  }
}
```

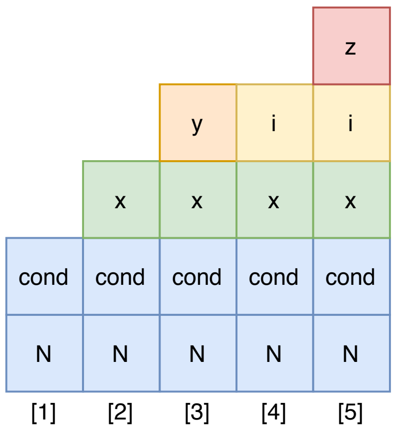


Figure 3.7: Graphical representation of variable scopes at different points in a program.

$$(\text{int})\ \frac{}{\Gamma \vdash n : \texttt{int}} \qquad \text{Assuming } n \text{ is an integer between } -2^{31} \text{ and } 2^{31}-1$$

$$(\text{add})\ \frac{\Gamma \vdash e_1 : T \qquad \Gamma \vdash e_2 : T \qquad T \in \{\texttt{byte}, \texttt{short}, \texttt{char}, \texttt{int}, \texttt{long}, \texttt{float}, \texttt{double}\}}{\Gamma \vdash e_1 + e_2 : T}$$

$$(\text{eq})\ \frac{\Gamma \vdash e_1 : T \qquad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1\ \texttt{==}\ e_2 : \texttt{boolean}}$$

Figure 3.8: Examples of typing rules rules for literals and expressions

variables or stack offsets—because different usages of the same identifier must always be resolved to the same variable. If the closest scope has an entry for some identifier then that variable will be used; otherwise, it recurses and asks its parent to perform a lookup. If the outermost scope still does not have a declaration for that identifier, then the compiler can safely reject the code on the basis that the programmer attempted to reference an undeclared variable. When registering a new variable, the `VariableScope` object checks whether it already has an entry for that identifier and throws an error if so, because variable shadowing is illegal in Java [12].

3.4.2 Type Checking

Java is a *statically-typed* language, which means that the type of each variable and expression can be determined at compile-time[1]. This enables the compiler to detect ill-typed programs and report an error, rather than generating incorrect WebAssembly code.

JavAssembler applies a typing judgement to every expression. For base literals, the typing judgement is simple: tokens like `5` and `1.5` will automatically be reported as `int` and `double` by the lexer. For every production rule in the grammar that allows expressions to be combined, there is a corresponding rule in the type system that states how to combine the types of the sub-expressions to determine the type of the combined expression. For example, an expression of the form e_1 `==` e_2 will be of type `boolean`, assuming that e_1 and e_2 both have the same type. This scheme allows the types of arbitrarily nested expressions to be determined. Examples of the typing rules used by JavAssembler are given in Figure 3.8.

Resolving the types of variables requires contextual knowledge: there is no way to infer the type of a variable `x` without having seen its declaration. Fortunately, the `VariableScope` class makes this easy, since variable types are registered with their containing scope when JavAssembler reads their declaration. Hence, types of variables can be looked up recursively just like their underlying variable or stack-offset allocations.

I implemented typing in JavAssembler by creating an interface called `Type`, and included a `getType()` method in the `Expression` interface. For the eight primitive types I created an enum, `PrimitiveType`, and for non-primitive types, I created an abstract class

[1]Technically, the *static type* of each expression can be determined at compile-time. The *dynamic type* of an expression may be any subtype of its static type.

`HeapObjectReference`, which `JavaClass` and `ItemArray` extend[2]. I chose an abstract class over an interface so that I could implement `getSize()` to return a constant 4 bytes. This setup made it easy to determine whether a given type is primitive just by using the `instanceof` operator.

JavAssembler performs type-checking in the constructor of each AST node data structure. All classes use the same `IncorrectTypeException` to indicate errors. For example, the `NotExpression` class, used to represent an application of the logical not operator (`!`), will throw this exception if the expression to which the negation is applied is not of type `boolean`. This gives the invariant that creating an ill-typed AST is impossible[3]. In addition, because type-checking is performed as part of the AST construction stage—while the parse tree is still being traversed—JavAssembler can report a precise error message stating the line and column at which the error occurred, without building a source map, using ANTLR's `ParserRuleContext` object.

Being able to extend classes is an important aspect of object-oriented programming. To support this, JavAssembler's type system includes a *subtyping* relation. If T is a subtype of T', then an expression of static type T may be used wherever an expression of type T' was expected. The immediate application of this is class inheritance. If a class `Child` extends `Parent`, then an instance of `Child` can validly be passed as an argument to a method expecting a parameter of type `Parent`. For variable assignments and function calls, rather than checking that the supplied type is the same as was declared for that variable or parameter, JavAssembler instead checks whether the supplied type is a subtype of it. To enforce this, I added the method `isSubtypeOf(Type other)` to the `Type` interface.

Performing these semantic checks early in the pipeline simplifies the work required by later stages since they are free to assume that the input program is legal and well-typed.

3.5 Language Features

So far, we have seen how JavAssembler processes an input file to generate an in-memory representation of the program, and how it checks the code of that program for correctness. The next challenge is to determine how to implement various high-level language features within the constraints of the WebAssembly runtime. I begin with the handling of object-oriented features like classes and inheritance, and later move on to method calls and dynamic polymorphism.

3.5.1 Classes and Inheritance

Classes are arguably the core component of object-oriented programming. A class essentially consists of two components: a set of attributes and a set of methods. Subclasses can add new attributes and methods, and may override existing methods, but crucially they

[2]The names `Class` and `Array` are reserved in Java.

[3]In normal Java, type errors may still occur at runtime if an invalid type cast is used. JavAssembler avoids this by not supporting casting!

```
class Parent {
    public int x;
    public double y;
}

class Child extends Parent {
    public float z;
}
```

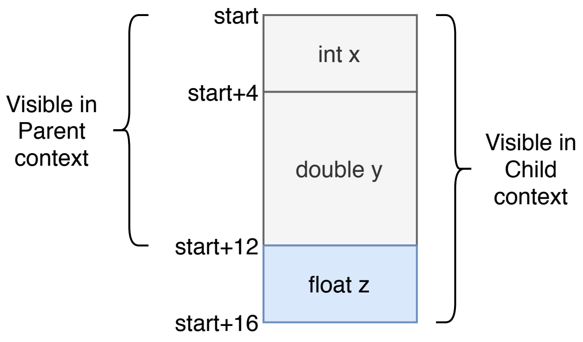


Figure 3.9: An illustration of how inheritance influences memory layouts.

cannot remove functionality of their parents. I designed the memory layout of objects using this principle.

The compiler needs to arrange objects in memory so that they can be accessed at runtime in a performance-efficient way. An advantage of the design I chose is that accesses to object attributes can always be resolved statically into reading from or writing to a memory address at a known offset from the start of the object, avoiding the need to perform complex address calculations at runtime.

Earlier, I mentioned that JavAssembler performs a topological sort of the class inheritance hierarchy to decide an order in which to process them. The reason for this is that JavAssembler always places new object attributes after those added in parent classes. This ensures that child classes still function correctly if used from the context of a parent class, since the offsets of the earlier attributes will be unchanged. Figure 3.9 shows an example. An added benefit is that JavAssembler can detect and report erroneous circular inheritance hierarchies.

To create an instance of a class, JavAssembler makes a call to `alloc_object`, which reserves space in the heap and returns a pointer to it. How this function works will be discussed in Section 3.6. If the object initialisation used a constructor, then JavAssembler statically looks up the function table entry of the constructor used and emits a call to it, setting up the arguments on the stack beforehand.

As well as storing the object's attributes, JavAssembler stores with each object a 12-byte header and variable-length footer. The header contains:

- A four byte `flags` field to be used by the garbage collector. While only two bits are ever used, profiling indicated that extending this to four bytes instead of one, the minimum addressable unit, increased attribute access performance by 4% due to the advantages of word-aligned address reads.[4]

- A `size` field to indicate the size of the object to the garbage collector.

- A virtual-table pointer.

The footer contains a bit for each four-byte word of the attributes region to indicate to the garbage collector whether to interpret that word as a pointer. Again for performance

[4]4% average increase in operations-per-second across the array and linked-list benchmarks in Section 4.3

reasons, the length of the footer is rounded up to the nearest four bytes. Placing the pointer information field in the footer rather than the header was not an arbitrary decision. Given that the size of the field depends on the number of attributes—which subclasses may increase—placing the pointer information in the footer was necessary to preserve the property that attributes can be looked up by reading at a constant offset relative to the start of an object, and that the offset can be determined with knowledge of only the static type of the object. This approach avoids having to perform address calculations at runtime. Figure 3.10 shows the memory layout that I implemented.

The topological sort ensures that classes are processed in an order consistent with their inheritance hierarchy, but it does not circumvent the problem that a class may still hold a reference of a type that has not yet been processed. A `Parent` class may validly have an attribute of type `Child`; a class may also hold a reference to an instance of the same type as itself, such as a recursive tree structure or a linked list. JavAssembler needs to process each class according to the topological sort so that child attributes can be laid out after the parent attributes, but also needs to validate that each attribute is of a valid type, which may not be possible at the time of processing that particular class if the relevant other type has not yet been processed. The solution I chose is to create the class `UnvalidatedJavaClassReference` that represents a reference to a type that is not currently in the class table. This is used for attributes as well as method parameters and return types. Once all classes have been processed, JavAssembler then performs another pass to replace all `UnvalidatedJavaClassReference` objects with the `JavaClass` object of the same name from the class table. This provides an opportunity to catch any references to undefined types and terminate with an appropriate error message.

3.5.2 Generic Types

Generic types enable the programmer to create classes that can hold attributes of arbitrary types. For example, the Java standard library provides a generic `HashSet<T>` class that programmers can use to store unordered sets of arbitrary types. An instance of the class cannot be created directly: rather, the generic class must be instantiated using type arguments, such as `HashSet<Integer>`. This alleviates what would otherwise be the arduous task of creating a different version of the class for each required type.

I implemented generic types in JavAssembler by creating the class `GenericJavaClass`. For each combination of type arguments used in the program, JavAssembler calls the `instantiate()` method, passing in a list of types, which builds a `JavaClass` object by replacing all `GenericType` instances in class attributes and method signatures. The class `GenericClassInstantiationCache` is used to cache classes so that instantiations using the same type arguments can be reused, saving memory. At compile-time, these distinct `JavaClass` objects are used so that expressions can be properly type-checked. However, Java forbids instantiating generic types with primitives [11], so only one version of each generic method is actually compiled, using a WebAssembly `i32` value for each generic type since all generic types will be resolved to heap objects.

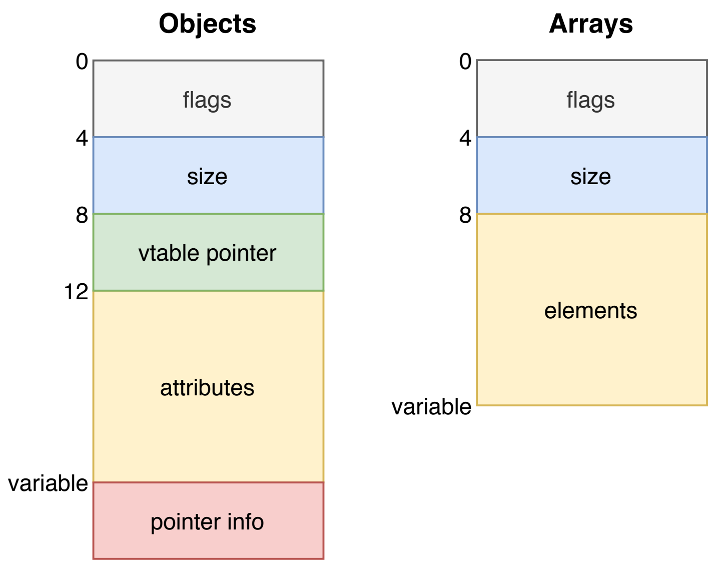


Figure 3.10: The memory layout of objects and arrays in the heap.

3.5.3 Arrays

Like classes, arrays are also exclusively heap-allocated in Java. This means that, like class attributes, the memory address at which a given element will be located cannot be known until runtime and must be expressed as an offset relative to the start of the array instead. The difference is that the index being accessed may also not be known until runtime—the address of `a[i]` depends on the value of `i`—so the offset also has to be computed at runtime and added to the start of the array.

To create an array, JavAssembler makes a call to `alloc_array`, which functions similarly to `alloc_object`. The main differences between the memory layouts of arrays and objects are that arrays do not need virtual-table pointers and do not need individual information about whether each element is a pointer: either all elements are pointers or none are, so a single-bit flag is sufficient. Figure 3.10 also shows the memory layout used for arrays.

To compute the heap address required to read or write to some index, JavAssembler first emits the code to compute the requested index, then multiplies this by the size of each element in bytes. This amounts to multiplication by a constant, because Java's static type system ensures that the type of the array will be known at compile-time. Finally, it adds this value to the address of the start of the array, accounting for the header, to identify the exact address to be used by the read or write operation. A bounds check is also applied to avoid accesses to invalid array addresses. Accesses to invalid addresses cannot in general be caught at compile-time, so instead a WebAssembly trap is used, terminating the program in the event that one occurs.

3.5.4 Static Method Calls

Another problem the compiler has to solve is determining where to jump to when a function is called. How this is implemented depends on whether the method is static.

```
public class Demo {
  [1] static int f();
  [2] static int f(int x);
  [3] static float f(float x);
  [4] static float g(float x);
  [5] static float g(float x, int y);
}
```

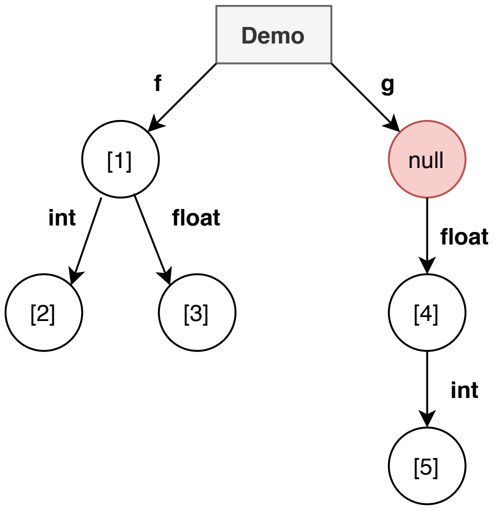


Figure 3.11: An example lookup tree.

Static methods are methods that exist independently from any object. They are what a non-OOP language might refer to as functions; the name comes from the fact that the address of the function being called can be determined at compile-time. In contrast, for non-static methods, different objects may have different implementations for the same method name and type signature, and the relevant method may not be knowable until runtime.

At first, the fact that WebAssembly supports symbolic function calling—that is, calling functions by name, such as in `call $myFunc`—may make this appear an easy task. However, Java complicates the problem by supporting *method overloading*, meaning that multiple methods with the same name may be defined within the same class, differing only in their argument types. Method resolution therefore requires more than just the name of the method: the types of arguments also must also be considered.

JavAssembler uses a class called `FunctionTable` to keep track of all functions in the program. In an initial pass over the source files, it registers each method with this table. The table can then be queried, returning a `FunctionTableEntry` object matching the requested name and type signature. If a method has been overloaded, then `FunctionTable` applies name mangling, appending the types of the arguments to the function's name. The returned `FunctionTableEntry` object will then contain the mangled name, which the code-generation stage uses to convert static Java method calls into unambiguous calls to uniquely-named WebAssembly functions.

For fast function table lookups, I created a generic acceleration structure called `LookupTree`. The function table uses this class to build a tree-like structure for each method name, meaning that stepping through the list of argument types is equivalent to walking down the tree to a particular entry, as illustrated in Figure 3.11. Using this structure, function resolutions can be performed in $\mathcal{O}(n)$ time where n is the length of the list of arguments.

```
public class Parent {
    public int doSomething() { ... }
}

public class Child extends Parent {
    @Override
    public int doSomething() { ... }
}

public class AnotherClass {
    public static int anotherMethod(Parent parent) {
        return parent.doSomething();
    }
}
```

Figure 3.12: An example of method overriding.

3.5.5 Dynamic Method Calls

Being able to extend classes and override methods is an important aspect of object-oriented programming, but does increase the complexity for compiler writers. Java's polymorphic behaviour means that, for a non-final class, knowledge of only the static type of an object is insufficient to work out which function will be used, because a subclass may override any non-final method. The actual function has to be determined at runtime. In the example in Figure 3.12, when the compiler compiles `anotherMethod`, there is no way to know which version of `doSomething` will be used.

The standard solution to this problem is to add a *virtual table* attribute to every object. A virtual table lists the function addresses of each non-static method in a class. Each class has its own virtual table, likely stored in the program's `data` section, which the virtual-table pointer points to. Rather than attempting to call a method directly, `anotherMethod` can instead perform an indirect call by dereferencing the virtual-table pointer of the passed `parent` object, calling the function in the first index. An advantage of this design is that child classes can easily append new methods to their virtual table without interfering with the methods expected of an instance of the parent class. While the static type of an object cannot be used to determine which function will be needed at runtime, it can be used to determine what the virtual-table offset of the method will be.

To implement this scheme, the compiler needs to build a virtual table for every class. JavAssembler builds these virtual tables at the same time as it builds the function table for static functions. A current WebAssembly restriction is that modules may contain only one table of functions [25], so JavAssembler instead concatenates all virtual tables and stores base offsets instead of pointers as virtual-table attributes. This transforms the problem of dynamic method resolution into one of resolving method references into virtual-table offsets so that an indirect call can be used through WebAssembly's `call_indirect` instruction. To perform these resolutions at compile-time, I reused the `LookupTree` structure, creating an instance for each method name in each class. The offset into the virtual table of a given method is determined by the static type of the object, guaranteeing that the index requested at runtime will always be valid.

3.6 Memory Management

So far, we have seen that JavAssembler supports heap objects by dynamically allocating memory from WebAssembly's linear memory space. This section discusses how I implemented memory allocation and garbage collection in JavAssembler.

3.6.1 Memory Layout Overview

As was discussed in Section 2.5.1, WebAssembly firmly separates its heap and call stack. This contrasts sharply with other instruction sets like x86, in which stack- and heap-allocated values reside within the same address space. The primary challenge of supporting garbage collection in WebAssembly is that the garbage collector needs the ability to inspect the whole stack, not just the current call frame, to determine which objects are reachable. If heap references were stored using WebAssembly local variables then this would not be possible.

The solution I used was to implement a second stack, stored within the linear memory space and managed manually, in which all pointers to heap objects are stored. This second stack is synchronised with the main WebAssembly call stack: call frames are manually pushed to and popped from it whenever functions are called and terminate. JavAssembler assigns primitive-typed local variables to local variables in the WebAssembly call stack, but assigns references to heap objects to call-frame offsets in the second stack. At no point is a heap object reference stored as a WebAssembly local variable, because this risks the object being garbage-collected while it remains accessible.

The garbage collection algorithm I chose to use is *Cheney's algorithm*, a form of copying collection. The idea is to partition the heap in half and copy accessible objects from the currently active half (called the *from-space*) to the inactive half (called the *to-space*) whenever garbage collection is required. The exact implementation will be discussed in Section 3.6.3; for now, the important part is that I needed to partition WebAssembly's linear memory in half and keep track of which side is active. The memory layout I implemented is shown in Figure 3.13.

Most compilers and ISAs use an upwards-growing heap and downwards-growing stack. I chose to reverse this in JavAssembler because WebAssembly's `load` and `store` instructions only allow positive offset arguments. Using an upwards-growing stack therefore allowed me to reduce the size of the generated code. Figure 3.14 shows this advantage, in this case when looking up a value at an offset of eight bytes.

I chose to reserve the address 0 to denote null pointers. When objects and arrays are dereferenced, JavAssembler emits code to test for zero and trigger a WebAssembly trap if true, effectively acting like a `NullPointerException`.

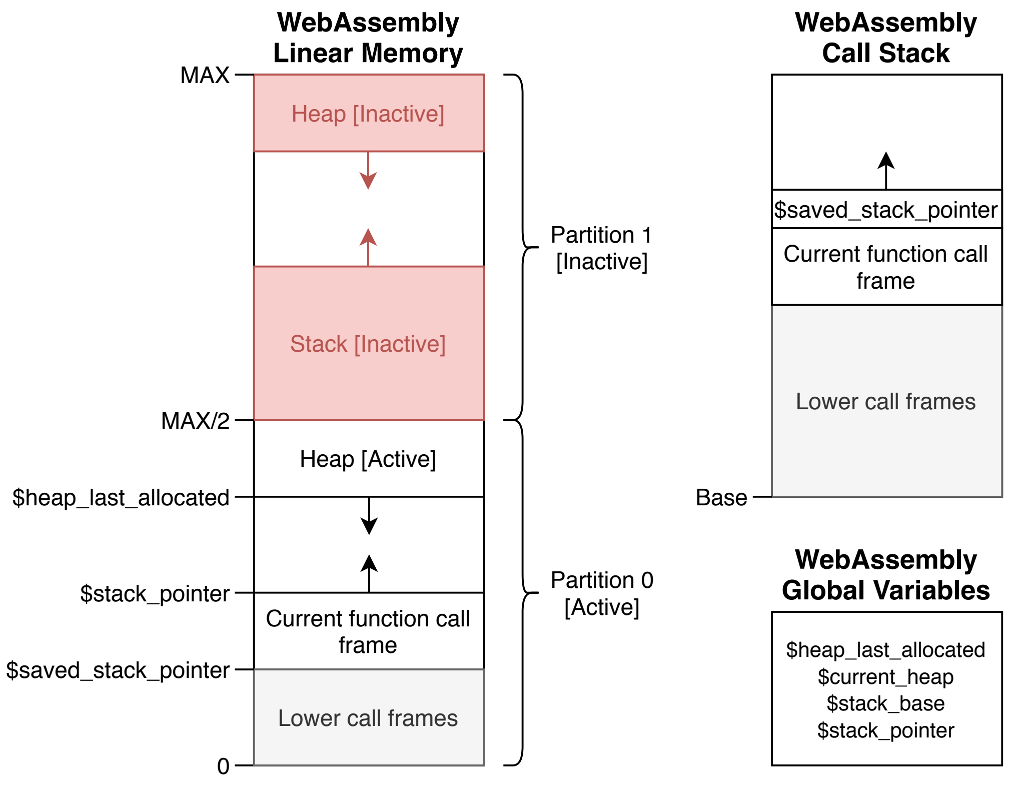


Figure 3.13: The memory layout used by JavAssembler.

```
global.get $call_frame_start
i32.const 8
i32.sub
i32.load
```

(a) Conventional downwards-growing stack

```
global.get $call_frame_start
i32.load offset=8
```

(b) JavAssembler's upwards-growing stack

Figure 3.14: The reduction in code size as a result of using an upwards-growing stack.

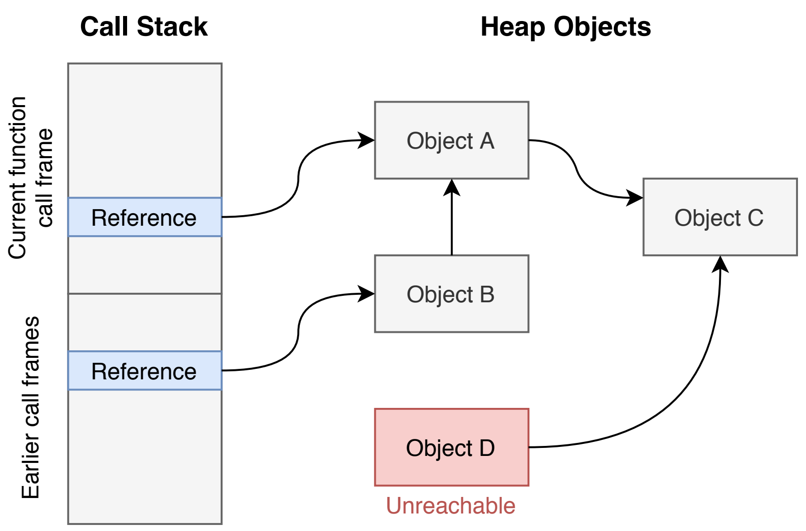


Figure 3.15: An example memory configuration before garbage collection.

3.6.2 Allocating Memory

With this layout, allocating memory at runtime is trivial. JavAssembler uses a *bump allocator*, which simply keeps a pointer to the last allocated address and decrements it whenever memory is requested. If there is insufficient free space between the heap and the stack, then the garbage collector is invoked. If there is still insufficient free space, then JavAssembler requests more memory from the JavaScript container.

3.6.3 Garbage Collection

Figure 3.15 gives an example of a situation in which garbage collection is required. For simplification, I have unified the two stacks. Objects A and B are directly accessible from the stack, so these need to be preserved. Object C is indirectly accessible, so that also needs to be copied. However, there is no way to access object D from the stack, so it can safely be deleted.

As discussed before, I chose to implement Cheney's algorithm because it uses $\mathcal{O}(1)$ additional space, meaning it will never cause a stack overflow. I wrote the garbage collector directly in WebAssembly, rather than compiling it from a higher-level language, so that I could integrate it exactly with the memory layout JavAssembler uses without having to deal with issues like endianness.

The algorithm essentially just performs a breadth-first search through the reachability graph, starting from the values directly accessible from the stack. In this example, objects A and B would be copied first, since these are the objects pointed to directly from the stack. Then, the algorithm moves on to consider all outward pointers from the objects it has just copied. Object A has a pointer to C, so C would be copied. The trick to the $\mathcal{O}(1)$ space overhead is that, rather than maintaining a queue like a typical breadth-first search algorithm, it merely needs two pointers into the to-space to indicate the objects yet to have their outward pointers traced. To avoid duplicating objects in the to-space, whenever the garbage collector moves an object across, it sets a flag on that object in

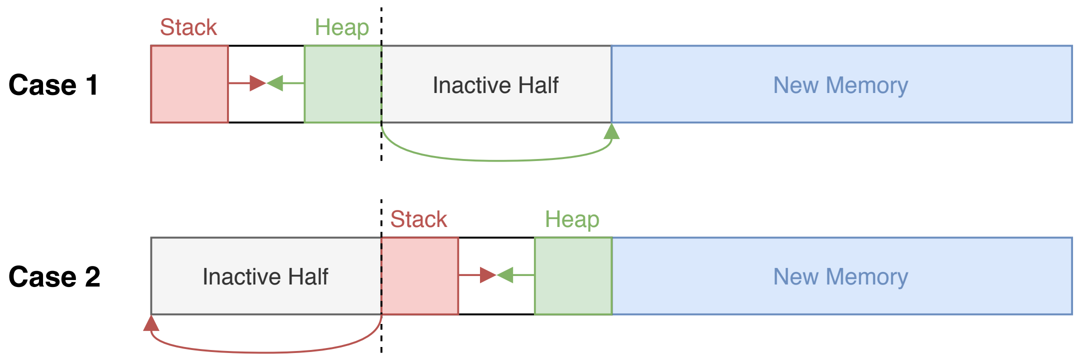


Figure 3.16: The two possible cases in a memory expansion.

the from-space and overwrites the `size` field with a pointer to the new location. Doing so avoids having to add an 'address copied to' field to each object, and means that the relocation of objects is idempotent.

The garbage collector disambiguates between arrays and objects by reading a flag. Objects contain a bit for each attribute to indicate whether to treat it as a pointer; for arrays, only one bit is needed since either all or none of the elements will be heap objects.

3.6.4 Memory Expansion

If the garbage collector fails to free enough space for the allocation, then the allocator requests more pages from the JavaScript host, shuffling the existing contents of the memory to fit the new layout. If the first half of the memory is currently in use (case 1), then this entails moving the heap and updating all pointers, including those in the stack. If the second half of the memory is in use (case 2), then only the stack has to be moved, with pointers unchanged. This is illustrated in Figure 3.16. Because moving the entire stack or heap is an expensive operation, I chose to double the size of the memory on each expansion.

3.7 Code Generation

The last stage of the compiler is the code-generation stage. This is where the in-memory representation of the program is finally translated to WebAssembly code and bundled with the memory-management subroutines I wrote. Here I explain the structure of JavAssembler's code-generation stage.

3.7.1 Emitting the WebAssembly Code

I began by writing a helper class to encapsulate writing to a file. I created the package `codegen` and the class `CodeEmitter`, which takes a file path as an argument to its constructor and manages opening and closing the file and writing to it via the public method `writeLine()`. One motivation for creating this class was that it could automatically handle indentation using the methods `increaseIndentationLevel()` and

Java Type	**Logical Size**	**WebAssembly Type**
`boolean`	1 bit	`i32`
`byte`	8 bits	`i32`*
`char`	16 bits	`i32`*
`short`	16 bits	`i32`*
`int`	32 bits	`i32`
`long`	64 bits	`i64`
`float`	32 bits	`f32`
`double`	64 bits	`f64`
Object reference	32 or 64 bits	`i32`

Table 3.2: Mappings from Java types to WebAssembly types. Asterisks denote that range-preservation code was required.

`decreaseIndentationLevel()`, enabling the rest of the code-generation stage to emit nicely formatted code.

The main entry point to the code-generation stage is the `WasmGenerator` class, which is responsible for emitting a WebAssembly module to a file. This includes emitting the concatenated virtual tables and setting up the linear memory.

As was discussed previously, JavAssembler uses the AST as its primary internal representation of the program; code-generation therefore amounts to a tree traversal. Impressed by the convenience of using the visitor pattern in the `ASTBuilder` class, I decided to reuse the design pattern in the code-generation stage too. I created the classes `ExpressionGenerator` and `StatementGenerator` to emit expressions and statements via the mutually recursive public methods `compileExpression()` and `compileStatement()`, which work by internally dispatching the argument to an appropriate private method.

3.7.2 Type Mappings

Java supports a richer set of types than WebAssembly. WebAssembly only supports 32- and 64-bit integers and floats, whereas the eight primitive types in Java span a far larger range of bit sizes and value ranges. The mapping I used to represent these types in WebAssembly is shown in Table 3.2.

When implementing the type mappings, I made sure not to permit any operations that would cause an out-of-range value to be stored. For example, a Java `short` type may take any integral value from -2^{15} to $2^{15}-1$, but the range of a signed WebAssembly `i32` type is -2^{31} to $2^{31}-1$. When applying a binary operator to a `byte`, `char` or `short` value, JavAssembler emits range-restriction code to ensure that the correct range is preserved. For `byte` and `short` values, which are signed, it applies a left-shift followed by a signed right-shift. `char` values are simpler because they are unsigned, so JavAssembler corrects their range just by using an `and` mask. Later, I wrote tests to validate that the overflow and underflow behaviour was correct for all primitive types.

Boolean values are represented using the `i32` constants 1 and 0 for `true` and `false`. I chose these values because they are closed under bitwise `and`, `or` and `xor` operations, and because WebAssembly interprets any non-zero integer as truthy when used as the condition for a conditional branch. WebAssembly does not provide a logical `not` operator, so JavAssembler instead computes it as an `xor` with the value 1.

3.7.3 Statements and Expressions

Though it does support local variables and provide heap access, WebAssembly is primarily a stack machine. Computation of expressions requires pushing values to the operand stack and consuming them with operations. For example, evaluating the expression $1 + 2$ could be performed by the instructions:

```
i32.const 1
i32.const 2
i32.add
```

Hence, to compute a binary operation over any arbitrary expressions, JavAssembler simply compiles each of the two sub-expressions, which will leave their values on the operand stack, then emits the relevant binary operator to consume them.

Function calls are performed by evaluating each of the arguments to leave them on the operand stack, then issuing a `call` instruction to call the relevant function. This is in accordance with Java's semantics, which state that arguments should be evaluated in left-to-right order [10]. The calling convention I used was that primitive function arguments are passed using the operand stack, and therefore consumed automatically by WebAssembly's function invocation procedure, while non-primitive arguments are manually copied to the new stack frame in the manually-managed second stack. Return values always use the main operand stack.

Assignments similarly require the creation of a value on the stack, followed by the consumption of that value using a `set` or `store` operation. To save a primitive value to a local variable, JavAssembler uses the `local.set` operation along with the index to which the variable was assigned. To reduce the code size I chose to use WebAssembly's numeric local variables, meaning that the generated WebAssembly code essentially uses *De Bruijn indices*. The indices are determined by the allocations made in the `VariableScope` class. Non-primitive values are written to the manual stack using a `store` instruction, such as `i32.store` for `i32` values, which also takes a memory address as an argument.

If-statements are easy to translate, because WebAssembly already provides an `if` construction. The condition expression is evaluated then consumed by the `if` instruction, which takes its branch only if the operand was non-zero. Else-branches are also supported. Java permits arbitrarily long chains of if-else-if statements, which JavAssembler handles by nesting the later if-statements within the else-branch of the first if-statement. To implement `for` and `while` loops, JavAssembler uses WebAssembly's `loop` construction. Rather than having to jump to a labelled program point, like in the x86 instruction set, WebAssembly's branches work as a stack: the operand to the `br` instruction is the number of 'levels' from which to exit. So, at the start of a while-loop, JavAssembler emits the code to test the condition, negates it and issues a conditional `br_if 1` statement to break

```
block
  loop
    ;; <evaluate expression>
    i32.const 1  ;; Negate the condition
    i32.xor
    br_if 1      ;; Conditionally exit the block

    ;; <loop body>

    br 0  ;; Unconditionally branch to loop start
  end
end
```

Figure 3.17: Example of a loop in WebAssembly.

from the loop if the condition is no longer true. Where other assembly languages may require an instruction like `jmp <address>` to restart the loop, WebAssembly simplifies this by offering the simple instruction `br 0`. An example of a loop in WebAssembly is shown in Figure 3.17.

Chapter 4

Evaluation

With the implementation of JavAssembler completed, the next stage was to evaluate it against its success criteria. This chapter begins with an explanation of the testing strategy I used, and moves on to explain how I used these tests to evaluate JavAssembler against its requirements. I then discuss the benchmarks I performed against reference implementations in JavaScript and C++, concluding with an analysis and explanation of the measurements.

4.1 Testing Strategy

While developing JavAssembler, I used JUnit [15] to write *unit tests*. A unit test validates the behaviour of one particular part of the program, such as a single method. For example, I used unit tests to test the `getSerialOrder()` method of the `TopologicalSort` class, each one applying the algorithm to a pre-prepared dependency graph and checking the result against the expected correct output. In total JavAssembler has 28 unit tests.

Unit tests are helpful for testing that individual components of the program work correctly, but are insufficient on their own for validating that an entire program is correct. *End-to-end* tests validate the entire output of a program. Being a compiler, this is a natural way to test that JavAssembler's output is correct, since the compilation process necessitates using the entire pipeline. I chose to write end-to-end tests using Jest [13], a JavaScript testing framework. The fact that WebAssembly is designed to be hosted within a JavaScript container made using a JavaScript testing framework the obvious choice. Jest provides simple assertion statements, such as `expect(...).toBe(...)`, as well as higher-level features, such as asserting that the invocation of a function will or will not cause an exception to be thrown. This was particularly useful for testing whether programs correctly trapped under certain illegal conditions, because WebAssembly traps are mapped to JavaScript exceptions.

Tests are written as JavaScript files with the `.test.js` suffix. For each test, I created a sample Java program exposing the relevant behaviour, and I wrote a script that uses JavAssembler to compile them all to WebAssembly. JavAssembler passes all 129 Jest test cases.

4.2 Success Criteria

In Section 2.1, I devised four main requirements of the project, two of which were deemed *core* tasks and the remaining two *extensions*. Here I discuss how I showed that the requirements have been satisfied.

- **(Core)** *Support compiling simple single-file programs consisting of stack variables and static methods*

  As I remarked in Section 2.1, demonstrating that I have achieved this criterion required the use of a significant proportion of the compiler. For each relevant language feature, I wrote a sample program that uses it, and produced Jest tests for typical and edge-case behaviours to ensure that the generated code adhered to Java's specifications.

  - I created `expressions.test.js` to test that expressions are parsed and evaluated correctly. I tested addition, subtraction, multiplication and division, including testing that integer division by zero triggers a WebAssembly trap and that floating-point division of a non-negative number by zero returns the JavaScript `Infinity` value. I tested that overflow and underflow behaviour works correctly for all five integral primitive types: for example, Figures 4.1 and 4.2 show how I validated overflow and underflow for Java's `int` type.

    Unfortunately, I was not able to validate Java's `long` type directly. Attempting to call a WebAssembly function that takes as an argument or returns an `i64` value from a JavaScript context causes an unconditional trap, because JavaScript 'does not currently have a precise way to represent these types' [9]. To work around this, I tested overflow and underflow behaviour for `long` values by writing Java functions that test the result themselves, returning a `boolean` value to indicate whether the test passed, then use Jest just to assert that the result is truthy, shown in Figures 4.3 and 4.4.
  - I tested language constructions like if-statements, while-loops and for-loops in `language_constructs.test.js`, for example by testing that loops iterated the expected number of times.
  - I tested that function-calling works in `functions.test.js`, including that arguments and return values are copied correctly.

  The fact that all of these tests pass shows that JavAssembler meets the first of its core requirements.

- **(Core)** *Support dynamic memory allocation for heap objects*

  In Section 2.1, I claimed that I could demonstrate working dynamic memory allocation by implementing a linked-list. To test it, I did exactly that: `dynamic_memory_allocation.test.js` includes a test that builds and traverses a linked-list, validating that the elements are stored correctly.

As well as supporting dynamically-allocated objects, JavAssembler exceeds the requirement by also supporting arrays. The tests for arrays are found in `arrays.test.js`. This file includes tests that arrays of both primitive and non-primitive types are stored correctly, and further that out-of-bounds array accesses trigger a WebAssembly trap. All of these tests pass, showing that JavAssembler meets the second of its core requirements.

- **(Extension)** *Support object-oriented programming features such as inheritance and dynamic polymorphism*

  As before, I verified that these had been implemented correctly by writing tests. The file `classes.test.js` contains tests for inheritance and dynamic polymorphism behaviour. I tested inheritance by defining methods and attributes in a parent class, then verifying that they can be accessed from a child class. I tested dynamic polymorphism by writing a method `isParent()` in the `Parent` class that always returns `true`, and then overriding it in the class `Child` with a method that always returns `false`. My tests confirmed that an instance of the `Child` class always uses the `Child` class's implementation and returns `false`, even when invoked from the static context of a `Parent`, showing that JavAssembler captured Java's dynamically polymorphic behaviour correctly. I actually managed to exceed these requirements by implementing generic types as well.

- **(Extension)** *Include garbage collection in the generated code*

  To test this, I created `garbage_collection.test.js`, which contains two tests. The first is a program that separately allocates 100,000 objects—more than can be simultaneously stored in the initial 64 KiB of memory—in such a way that each one goes out of scope after being created, hence leaving it available for deletion by the garbage collector. The test monitors memory usage to make sure that the generated code does not request additional memory from the JavaScript host environment.

  I separately tested the correctness of the memory expansion system by writing a test that reserves memory for an array of length 100,000 and populates it with 100,000 dynamically-allocated `Integer` objects. Being larger than the default memory size, the system has to request more memory from the host environment. In addition to initially requesting space for the array, it also has to allocate memory continually for each `Integer` object. The test then loops through the array once more to validate that the values are stored correctly and in valid memory addresses.

These tests show that JavAssembler meets all of the functional requirements set out for it. Instructions for how to run the tests can be found in the file `README.md`.

```java
public class TypeRanges {
    public static int addInts(int a, int b) {
        return a + b;
    }
    public static int subtractInts(int a, int b) {
        return a - b;
    }
    ...
}
```

Figure 4.1: The Java code used to test addition and subtraction of integers.

```javascript
describe('Integers', () => {
  test('Overflow is correct', () => {
    const result = wasmInstance.TypeRanges_addInts(Math.pow(2,31)-1, 1);
    expect(result).toBe(-Math.pow(2,31));
  });
  test('Underflow is correct', () => {
    const result = wasmInstance.TypeRanges_subtractInts(
            -Math.pow(2,31), 1);
    expect(result).toBe(Math.pow(2,31)-1);
  });
});
```

Figure 4.2: The Jest scripts I used for testing the overflow behaviour of the Java functions defined in Figure 4.1.

```java
public static boolean testLongOverflow() {
    long sum = 9223372036854775807L + 1L;
    return sum == -9223372036854775808L;
}
```

Figure 4.3: The method used to test overflow behaviour for Java's `long` type.

```javascript
describe('Longs', () => {
  test('Overflow is correct', () => {
    const result = wasmInstance.TypeRanges_testLongOverflow();
    expect(result).toBeTruthy();
  })
  ...
})
```

Figure 4.4: The Jest code for validating the test in Figure 4.3.

4.3 Benchmarking

The tests showed that JavAssembler produces valid WebAssembly for every supported language feature. However, I wanted to determine not only whether writing such a compiler is possible: I wanted to find out whether JavAssembler actually offers a speedup over existing technologies. For this, I needed to use *benchmarks*.

I benchmarked JavAssembler by constructing four sample programs designed to cover different features of Java. I engineered all benchmarks to include some form of difficulty parameter that would enable me to measure how execution time scales with respect to the problem size. The benchmarks I used are:

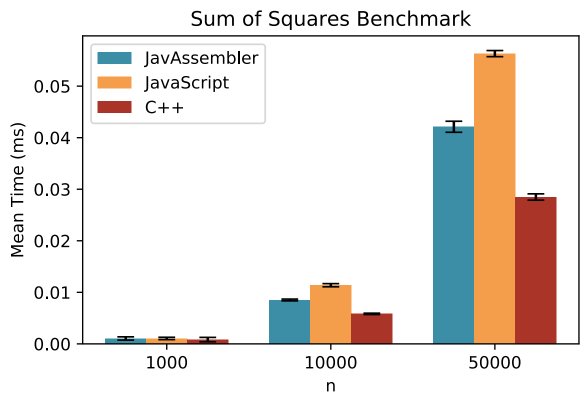
- **Sum-of-Squares** This benchmark computes the sum of the first n square numbers. I selected it to test general looping and arithmetic performance.
- **Recursion** This benchmark involves a recursive function calling itself a specified number of times. The main incentive was to test how much the manual stack management affected function-calling performance in JavAssembler.
- **Linked-List Traversal** This benchmark involves creating a linked-list of n distinct integers and iterating through it. As well as testing the random memory-access performance, it also tests the performance of the memory allocator and garbage collector. Not every run will trigger the garbage collector, but `benchmark.js` addresses this by repeating the benchmark many times and averaging the result.
- **Array Traversal** This benchmark involves creating an array of size n, populating it with the numbers 0 to $n-1$, then iterating over every element of it. I designed this benchmark to test the sequential memory-access performance of JavAssembler.

I implemented all of these benchmarks in Java and compiled them to WebAssembly using JavAssembler. Then, I created two reference implementations of each benchmark in JavaScript and C++, being careful to keep the implementations as similar as possible. I chose to benchmark against JavaScript's performance because improving the performance of web applications is one of WebAssembly's reasons for existence. I chose C++ because it is currently one of the most popular languages for producing WebAssembly libraries. I compiled the C++ code using Emscripten, disabling optimisations to prevent the benchmarks from being optimised away by dead-code elimination. The file `README.md` contains instructions for how to reproduce the benchmarks and generate the graphs.

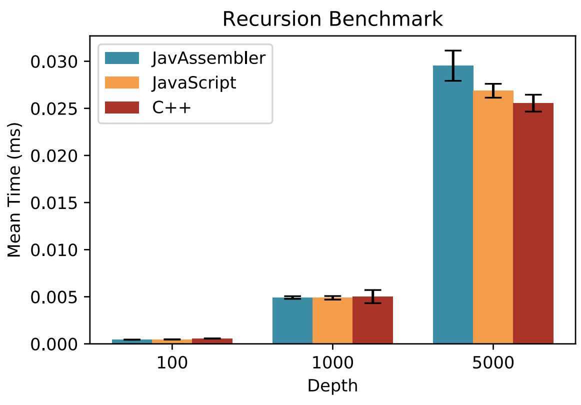
4.3.1 Results and Analysis

The full benchmarking results are given in Appendix A, and have been plotted in Figure 4.5.

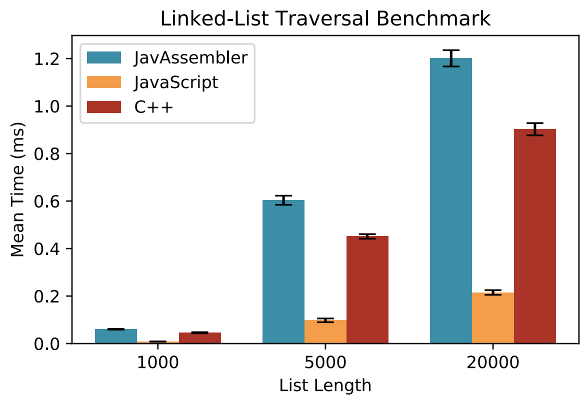
JavAssembler's output performed well in the sum-of-squares benchmark, surpassing the performance of JavaScript in all three cases and not falling too far behind the C++/Emscripten reference. It was the slowest of the three in the recursion benchmark, but only took 16% longer than C++, the fastest performer. These results show



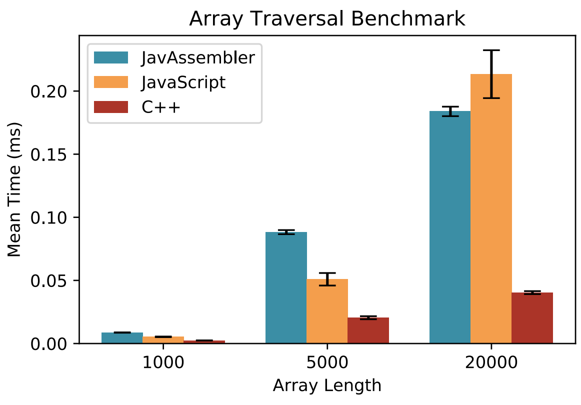
(a) Sum of squares benchmark



(b) Recursion benchmark

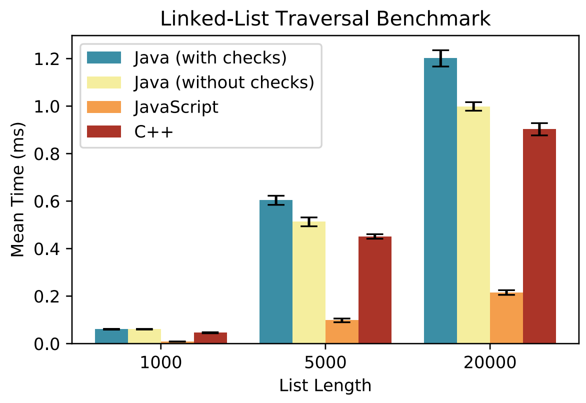


(c) Linked-list traversal benchmark

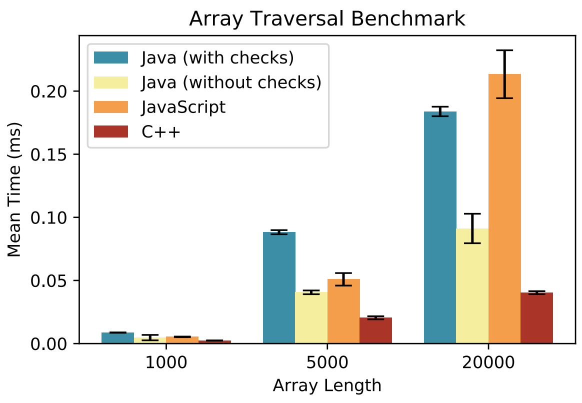


(d) Array-traversal benchmark

Figure 4.5: Benchmarking results. Error bars show the standard deviation of each sample.



(a) Linked-list traversal benchmark



(b) Array traversal benchmark

Figure 4.6: Repeated benchmarks with null pointer-checking and array bounds-checking disabled.

that WebAssembly certainly does offer a speedup over JavaScript, at least in certain computationally-bound tasks.

The surprise winner in the linked-list benchmark was the JavaScript implementation, not C++ as I had expected given the results so far. Profiling using the `node --prof` command revealed that the JavAssembler implementation spent 7.1% of its time executing the `set_at_stack_frame_offset` function alone, showing just how much of an impact the second stack has on execution time. If WebAssembly were to offer built-in garbage collection, then these overheads would not be required. 19.2% of the time was spent directly in the `LinkedList_append` function, which contains three checks for null pointers. Most of the remaining time was spent in various memory-management functions. Having to manage the stack manually, rather than leaving it to the V8 runtime, is the most probable reason that both WebAssembly implementations failed to beat the JavaScript implementation.

The WebAssembly environments fared better in the array-traversal benchmark, although the C++ reference outperformed JavAssembler decisively. The C++ implementation averaged only 0.04 ms where JavAssembler's generated code took 0.18 ms.

Java's use of runtime null-pointer checking and array bounds-checking puts it at an inherent performance disadvantage compared with C++, which instead considers dereferencing of invalid pointers and out-of-bounds array indices merely to be *undefined behaviour*. To determine the extent to which these checks impacted the performance of JavAssembler, I re-ran the linked-list and array-traversal benchmarks with null pointer and array bounds checks disabled. The results are graphed in Figure 4.6.

The effect on the array-traversal benchmark was dramatic: removing bounds-checks cut the execution time in half. Runtime bounds-checking clearly causes a significant performance penalty. The change to the linked-list benchmark was a less significant but nonetheless measurable improvement, showing that null pointer checks do harm performance slightly. These measurements show that Java is intrinsically a slower language than C++, and performance parity would never have been expected. Unfortunately, however, JavAssembler still has to duplicate work in managing two stacks and manually copying function arguments, rather than being able to let the underlying V8 runtime handle it in a more unified fashion. This is a result of WebAssembly's design, not Java. For this reason, I firmly support the proposal to add garbage collection to WebAssembly, including the addition of reference types to the type system [24].

4.4 Summary

JavAssembler met or exceeded each of the four main requirements originally set out, evidenced by tests included with the repository. For this reason, I deem the project to have been a success. Furthermore, its favourable performance in the sum-of-squares and array-traversal benchmarks compared with JavaScript demonstrates that WebAssembly clearly has the potential to improve the performance of current and future web applications, even if its lack of object-awareness holds back its performance for some languages. The project has convinced me that WebAssembly is an important new language that is likely to have a significant impact on future web application development.

Chapter 5

Conclusions

On a personal level, this is the largest software project I have ever worked on. I experienced first-hand the advantages of writing clear and well-documented code, and benefited directly from using a test-driven development cycle. My deeper knowledge of programming languages and compilers will no doubt be hugely beneficial to my future career as a software developer.

JavAssembler provably met the requirements set out for it, and I am satisfied with the performance attained. WebAssembly clearly offers lots of potential for future web applications, and source languages like Java offer a middle-ground between the performance of a lower-level language like C++ and the conveniences of features like automated memory management. If the garbage-collection proposal is accepted, then compilers like JavAssembler will be able to produce code that runs far more naturally in a WebAssembly environment, and hence is far more amenable to optimisation by V8's JIT compiler.

A possible future extension would be to support more of Java's language features, such as interfaces, enums and lambda functions. It would also be possible to use the second stack to overcome WebAssembly's current lack of support for exception handling. Low-level optimisations like data-flow analysis would be of limited utility because WebAssembly runtimes like V8 tend to apply them anyway, but adding higher-level optimisations like function inlining would certainly be viable. With more time to implement the compiler, I would have liked to have used a longer pipeline, separating the AST-construction and semantic-analysis stages for improved modularity. Overall, however, I am pleased with the project and with the results that JavAssembler was able to produce.

WebAssembly marks an exciting new era for web technologies. I believe that the rise of mobile- and web-based applications, combined with the decline of Moore's law, will place an ever larger value on software being faster and more efficient. JavaScript changed dramatically during the 2010s: perhaps the 2020s will be the decade of WebAssembly.

Bibliography

[1] An Introduction to Speculative Optimization in V8. `https://ponyfoo.com/articles/an-introduction-to-speculative-optimization-in-v8`. Accessed: 27 March 2020.

[2] Android Open Source Project Java Code Style for Contributors. `https://source.android.com/setup/contribute/code-style`. Accessed: 19 March 2020.

[3] ANTLR: ANother Tool for Language Recognition. `https://www.antlr.org/`. Accessed: 19 March 2020.

[4] Apache Commons CLI Library. `http://commons.apache.org/proper/commons-cli/`. Accessed: 19 March 2020.

[5] asm.js. `asmjs.org`. Accessed: 3 March 2020.

[6] Benchmark.js: A benchmarking library that supports high-resolution timers and returns statistically significant results. `https://benchmarkjs.com/`. Accessed: 28 March 2020.

[7] Emscripten. `https://emscripten.org/`. Accessed: 17 March 2020.

[8] Emscripten: WebAssembly Standalone. `https://github.com/emscripten-core/emscripten/wiki/WebAssembly-Standalone`. Accessed: 17 March 2020.

[9] Exported webassembly functions. `https://developer.mozilla.org/en-US/docs/WebAssembly/Exported_functions`. Accessed: 9 April 2020.

[10] Java Language Specification: Expressions. `https://docs.oracle.com/javase/specs/jls/se7/html/jls-15.html`. Accessed: 14 April 2020.

[11] Java Language Specification: Generics. `https://docs.oracle.com/javase/tutorial/java/generics/restrictions.html`. Accessed: 15 April 2020.

[12] Java Language Specification: Names. `https://docs.oracle.com/javase/specs/jls/se7/html/jls-6.html#jls-6.4`. Accessed: 15 April 2020.

[13] Jest: Delightful JavaScript Testing. `https://jestjs.io/`. Accessed: 28 March 2020.

[14] JLang: An LLVM backend for the Polyglot compiler. `https://polyglot-compiler.github.io/JLang/`. Accessed: 3 March 2020.

[15] JUnit 5. `https://junit.org/junit5/`. Accessed: 14 April 2020.

[16] JWebAssembly: A Java bytecode to WebAssembly compiler. `https://github.com/i-net-software/JWebAssembly`. Accessed: 28 March 2020.

[17] Mockito: Tasty mocking framework for unit tests in Java. `https://site.mockito.org/`. Accessed: 14 April 2020.

[18] Polyglot: An open compiler front end framework for building Java language extensions. `https://www.cs.cornell.edu/projects/polyglot/`. Accessed: 3 March 2020.

[19] TIOBE Index for February 2020. `https://www.tiobe.com/tiobe-index/`. Accessed: 3 March 2020.

[20] V8 JavaScript Engine. `v8.dev`. Accessed: 27 March 2020.

[21] WebAssembly. `https://webassembly.org/`. Accessed: 3 March 2020.

[22] WebAssembly 1.0 Becomes a W3C Recommendation and the Fourth Language to Run Natively in Browsers. `https://www.infoq.com/news/2019/12/webassembly-w3c-recommendation/`. Accessed: 3 March 2020.

[23] WebAssembly Binary Toolkit. `https://github.com/WebAssembly/wabt`. Accessed: 18 March 2020.

[24] WebAssembly: Features to add after the MVP. `https://webassembly.org/docs/future-features/`. Accessed: 19 April 2020.

[25] WebAssembly Specifications: Modules. `https://webassembly.github.io/spec/core/syntax/modules.html`. Accessed: 17 April 2020.

[26] C. J. Cheney. A nonrecursive list compacting algorithm. *Commun. ACM*, 13(11):677–678, November 1970.

[27] Abhinav Jangda, Bobby Powers, Arjun Guha, and Emery Berger. Mind the gap: Analyzing the performance of webassembly vs. native code. *CoRR*, abs/1901.09056, 2019.

[28] R. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008.

Appendix A

Benchmarking Results

Asterisks denote re-runs with null pointer-checking and array bounds-checking disabled.

Benchmark	**Environment**	**Size**	**Mean (ms)**	**Std. Dev (ms)**
SumSquares	JavAssembler	1000	0.001049	0.000317
SumSquares	JavaScript	1000	0.001053	0.000226
SumSquares	C++	1000	0.000834	0.000423
SumSquares	JavAssembler	10000	0.008515	0.000174
SumSquares	JavaScript	10000	0.011381	0.000304
SumSquares	C++	10000	0.005822	0.000132
SumSquares	JavAssembler	50000	0.042118	0.001072
SumSquares	JavaScript	50000	0.056310	0.000593
SumSquares	C++	50000	0.028491	0.000615
Recursion	JavAssembler	100	0.000443	0.000011
Recursion	JavaScript	100	0.000470	0.000016
Recursion	C++	100	0.000569	0.000014
Recursion	JavAssembler	1000	0.004916	0.000142
Recursion	JavaScript	1000	0.004887	0.000189
Recursion	C++	1000	0.005015	0.000698
Recursion	JavAssembler	5000	0.029529	0.001605
Recursion	JavaScript	5000	0.026869	0.000736
Recursion	C++	5000	0.025551	0.000897
LinkedList	JavAssembler	1000	0.060407	0.001707
LinkedList	JavAssembler*	1000	0.060517	0.018016
LinkedList	JavaScript	1000	0.008518	0.000460
LinkedList	C++	1000	0.045671	0.002241
LinkedList	JavAssembler	10000	0.603376	0.019181
LinkedList	JavAssembler*	1000	0.512638	0.019000
LinkedList	JavaScript	10000	0.097731	0.007885
LinkedList	C++	10000	0.451093	0.009589
LinkedList	JavAssembler	20000	1.200958	0.034301
LinkedList	JavAssembler*	20000	0.998568	0.018016
LinkedList	JavaScript	20000	0.215071	0.009854
LinkedList	C++	20000	0.902500	0.025806
Array	JavAssembler	1000	0.008702	0.000149

Benchmark	**Environment**	**Size**	**Mean (ms)**	**Std. Dev (ms)**
Array	JavAssembler*	1000	0.004720	0.002134
Array	JavaScript	1000	0.005309	0.000250
Array	C++	1000	0.002428	0.000080
Array	JavAssembler	10000	0.088255	0.001632
Array	JavAssembler*	10000	0.040593	0.001485
Array	JavaScript	10000	0.050906	0.004960
Array	C++	10000	0.020441	0.001211
Array	JavAssembler	20000	0.183881	0.003764
Array	JavAssembler*	20000	0.091174	0.011670
Array	JavaScript	20000	0.213393	0.018982
Array	C++	20000	0.040325	0.001195

Project Proposal: Implementing a Java to WebAssembly Compiler

Michaelmas 2019

Project Originator	Dr Timothy M. Jones
Project Supervisor	Dr Timothy M. Jones
Director of Studies	Professor Lawrence C. Paulson

1 Description

The goal of the project is to write a compiler for a subset of Java, targeting WebAssembly. I plan to use Java as the implementation language.

WebAssembly (abbreviated *Wasm*) is a binary instruction format for a stack-based virtual machine [4]. The WebAssembly instruction format resembles other low-level assembly formats, so the generated code is highly amenable to ahead-of-time optimisation. The instruction set is low-level: for example, at the time of writing, WebAssembly does not provide garbage collection [5].

One of the motivations for WebAssembly's development was the poor performance of JavaScript for intensive computations. Being an interpreted and dynamically typed language makes ahead-of-time compilation for JavaScript very challenging. WebAssembly was introduced by the World Wide Web Consortium to allow developers to write performance-integral parts of their code in a compiled language, then bundle the compiled code with the rest of their application in a format inter-operable with JavaScript.

A compiler for Java will be interesting because WebAssembly was originally designed as a target for languages like C, C++ and Rust [4]. These languages require manual memory management: when a region of memory that was dynamically allocated for an object is no longer required, the programmer is responsible for ensuring that the region is deallocated (for example, by using a `free()` function or `delete` operator). Compiling from Java will be harder because the language permits the programmer to write code under the assumption that a garbage collector will be provided. Java does not even have a construction for deleting objects. This poses a challenge because WebAssembly has no garbage collector, so I will have to implement garbage collection myself.

2 Starting Point

WebAssembly is far from the first attempt to improve the performance of web applications. `asm.js` designed to be an easily optimisable, low-level subset of JavaScript that compilers can target from higher-level languages like C [2]. The resulting JavaScript code can be executed with only a factor of 2 slowdown compared with natively compiled C [3]. However, WebAssembly can be parsed an order of magnitude faster than `asm.js` and takes advantage of more CPU features, leading to performance gains as significant as 4× faster operations on 64-bit integers when compared with `asm.js` [6]. This is why I have chosen to target WebAssembly as the output of the compiler.

Rather than hand-crafting a lexer and a parser, I intend to use ANTLR [1], a parser generator. Generated parsers tend to be more maintainable than manually written ones: to add parser support for a new feature of the language, I would only have to modify the grammar. A manually written parser would be needlessly complicated to extend and maintain.

3 Structure and Implementation of the Project

- The *front-end* of the compiler deals with *lexing* and *parsing* the source code – the aim is to output an *abstract syntax tree* (AST). As mentioned previously, I do not intend to completely write this stage myself: I will use ANTLR to generate a parser. To do this, I need to specify a grammar to recognise `.java` files, and let ANTLR derive the exact parsing logic. Using this workflow means it will be easy to make changes to the grammar should I wish to support new features in the future.

- At the *back-end* of the compiler is the *code-generation* stage. This is where the actual WebAssembly code will be produced. The first step is to read the official WebAssembly binary-format specification and decide how to handle different Java constructs. As an example, `while` loops will need to be converted to a test and a set of jumps. Then, I will need to implement code-generation routines for each supported language construct, writing them to an output file.

- Larger software projects are typically split into multiple files. I will have to implement a *linker* so that the compiler can resolve names defined in other files, through the use of a symbol table.

- Java is an object-oriented language, and *inheritance* is a central concept in object-oriented programming. According to its specification, Java uses *dynamic polymorphism*, which means that method resolution can only be computed at runtime. To solve this, dynamically polymorphic OOP languages typically use a *virtual table* ("*vtable*") for method resolution. A virtual table contains function addresses for each method in a class. There will be one virtual table in memory for each class loaded. Every object has a reference to its virtual table: this reference can be used to look up the address to jump to when calling methods on an object. A subtlety in WebAssembly is that functions are not referenced by pointers; rather, they are referenced by their index in a module-wide function table.

- I will need to implement a garbage collector to ensure that reserved memory that is no longer in use can be recycled. I will have to decide a strategy for tagging variables so that the garbage collector knows whether to follow them as pointers. If time permits, I would also like to investigate different strategies for deciding when to invoke garbage collection - for example, running the garbage collector preemptively may give better performance compared with waiting for the heap to run out of free space. Again if time permits, I would like to investigate the performance difference between algorithms like copying collection and mark-and-sweep.

4 Success Criteria

The goal of this project is to compile a program written in a subset of Java to WebAssembly and have it execute correctly.

I intend to measure the degree to which code generated by a simple compiler like this is able to outperform a like-for-like implementation of the same algorithm written directly in JavaScript. It will be interesting to see whether a highly optimised JavaScript runtime like Chrome's V8 engine is able to overcome the inherent slowness of using an interpreted language.

As a minimum, the compiler should support simple programs written in a single `.java` file consisting of only stack variables and static functions. Ideally, it would also support dynamic memory allocation for heap objects. It would also be preferable if the compiler could support object-oriented programming features such as inheritance and polymorphism. Finally, the generated code would ideally contain some form of garbage collection. If these goals are achieved, then I will deem the project to have been a success.

Possible extensions include experimenting with different garbage collection algorithms (for example, mark-and-sweep vs copying collection) and different strategies for deciding when to invoke garbage collection. If time permits, a further extension would be to implement a simple code optimisation stage.

5 Plan of Work

Following is a plan of the work to be completed, separated into ten blocks, each of which is expected to require around two weeks to complete.

Block 1 – Due 4/11/2019

- Create the project in IntelliJ

- Set up a GitHub repository to host the source code
- Set up Gradle and add ANTLR as a dependency
- Write a grammar to recognise Java files

Block 2 – Due 18/11/2019

- Research file layout for WebAssembly modules and memory layout for loaded WebAssembly modules
- Implement code-generation stage for proof-of-concept subset of the language
- Write a JavaScript program to test generated code
- Note: I have a deadline for Digital Signal Processing on 15/11/2019

Milestone

At this stage, the compiler should support compiling small programs correctly. For example, variable declarations, assignments and manipulations should be supported, as should flow control structures like branches (`if` and `else`) and loops (`for` and `while`).

Block 3 – Due 2/12/2019

- Implement a linker to support compiling multiple interdependent files at once
- Implement a memory allocator to make the heap usable
- Update the grammar to support the `new` keyword and generate a new parser
- Implement dynamically creating objects, permitting reading and writing their public attributes

Milestone

It should now be possible to dynamically initialise objects. This can be demonstrated by compiling a linked list implementation that relies on allocating memory for nodes at runtime.

Block 4 – Due 16/12/2019

- Implement generation of a virtual table for each class
- Support calling methods on objects by looking up the function index from the virtual table at runtime

Milestone

It should now be possible to create objects and call their methods. A way to validate that dynamic polymorphism is working correctly would be to define a function in one class that outputs a constant, then override it in a subclass with a function that outputs a different constant, and then call the method from a polymorphic context.

Block 5 – Due 6/1/2020

- Determine a strategy for tagging variables to indicate whether they are a pointer
- Implement a copying garbage collection algorithm
- Modify the memory allocator to invoke the garbage collector if memory has been requested but there is not enough free space in the heap

Milestone

The core implementation of the compiler should now be complete. To test that garbage collection is working correctly, I can write a program that repeatedly allocates objects, then lets their reference go out of scope. If garbage collection is working correctly then the program should only require a bounded amount of memory, regardless of how long it executes for.

Block 6 – Due 20/1/2020

- Implement a mark-and-sweep garbage collection algorithm
- Profile the results using different example programs to determine which algorithm gives better performance
- If time permits, add a simple optimisation stage (for example, remove empty `else`-blocks)

Block 7 – Due 3/2/2020

- Testing and bug fixing

Block 8 – Due 17/2/2020

- Create the document for the dissertation in Overleaf
- Set up a GitHub repository for tracking changes to the document
- Write the *introduction* section
- Write the *preparation* section

Block 9 – Due 2/3/2020

- Write the *implementation* section

Block 10 – Due 11/3/2020

- Write the *evaluation* section
- Write the *conclusion* section

Milestone

The project should now be complete, and a first draft of the dissertation should be ready for proof-reading.

6 Resource Declaration

I intend to work on the project using my personal laptop. I plan to use IntelliJ IDEA Ultimate as my IDE, and LaTeX and Overleaf for typesetting the dissertation. My laptop is a 2017 MacBook Pro with an Intel Core i5 processor, 16GB of RAM and a 256GB SSD. I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure.

As well as keeping a local Git repository for version control, I will also push commits to a private GitHub repository. In addition, I will save weekly backups of both the code and the dissertation to my personal Google Drive account. As a student of the University I will have access to IntelliJ IDEA Ultimate for the duration of the project. In the case that this arrangement is discontinued, I will instead use another text editor, such as Sublime Text. In the event of the loss of or damage to my laptop, I will continue the project using the MCS machines.

References

[1] Antlr (another tool for language recognition). `https://www.antlr.org/`. Accessed: 13 October 2019.

[2] asm.js. `asmjs.org`. Accessed: 16 October 2019.

[3] asm.js frequenty asked questions. `http://asmjs.org/faq.html`. Accessed: 16 October 2019.

[4] Webassembly. `https://webassembly.org/`. Accessed: 14 October 2019.

[5] Webassembly: Features to add after the mvp. `https://webassembly.org/docs/future-features/`. Accessed: 14 October 2019.

[6] A. Zakai. Why webassembly is faster than asm.js. `https://hacks.mozilla.org/2017/03/why-webassembly-is-faster-than-asm-js/`. Accessed: 16 October 2019.