Bundesrealgymnasium Feldkirchen

Flurweg 3

9560 Feldkirchen

Implementing “wrk”

Discussing the Design and Implementation of an Experimental Programming Language

Leo Gaskin, 8B

19th February, 2020

supervised by

Mag. Manuel Benjamin-Leitner

Abstract

Programming languages and compilers have long stood at the forefront of innovation in computing. Techniques and lessons relating to the implementation of compilers or more generally language processors still remain highly relevant today, as new technologies allow them to be applied to entirely new domains of computing, such as the modern web. This paper presents “wrk”, an experimental programming language, targeting the emerging WebAssembly technology, and its reference compiler, implemented using the Rust programming language. We give an overview of both design and implementation, then focus on some relevant unconventional choices, where possible providing appropriate rationale for their inclusion. The paper concludes by evaluating these findings and their applications while also giving a brief outlook on possible future developments.

Contents

1 Introduction

This paper discusses the design and implementation of programming languages, as they relate to the domains of compilers and computer science. This is achieved by describing a new experimental programming language, dubbed “wrk”, which has been developed specifically for examination in this paper. The primary aim is to present a complete and usable programming language along with a working compiler, that make use of principles of modern research into their fields. Secondarily, the paper also provides a comparatively simple introduction into these topics, which should make them understandable to a non-technical audience.

As such, the structure follows the obvious pattern of first introducing the central topic of compiler design, then applying these learned concepts to the design and implementation of the “wrk” language and compiler. At last, partly through the examination of an exemplary application implemented in “wrk”, the results of this original research are evaluated and an outlook on possible future developments is given.

In doing so, we primarily source well-known introductory texts on the topic of compiler design, but also make use of newer research into existing programming languages and texts relating to specific technologies this paper relies on.

2 Introduction to Compiler Theory

> Programming languages are notations for describing computations to people and to machines. The world as we know it depends on programming languages, because all the software running on all the computers was written in some programming language. But, before a program can be run, it first must be translated in to a form in which it can be executed by a computer. The software systems that do this translation are called compilers.[1]

Because compilers are this integral to the domain of computer programming a lot of time and effort has been invested into researching the many different aspects of these applications. In turn, compilers and other tools in the area of programming languages play a central role in providing new ways of abstraction for developers.[2] They are often at the forefront of innovations in computing and act as a testing ground for new ideas and concepts in the fields of computer science and software engineering.[3]

For the inherent complexity of the issues met in designing a compiler, numerous techniques and principles intended to simplify this process, many of which also prove to be useful for other classes of applications, have developed.[4] However, as modern compilers themselves are often based on large and complex codebases, traditional principles of software engineering should generally also be applied to keep them maintainable and useful.[5]

Understanding at least some of these concepts is generally speaking a prerequisite to understanding any compiler and the choices made in its design and implementation.

[1]Aho et al. 2007, p. 1.
[2]Cf. ibid., p. 15.
[3]Cf. Abelson and Sussman 1996, p. 332.
[4]Cf. Aho et al. 2007, p. 1.
[5]Cf. Cooper and Torczon 2011, p. 1.

2.1 Classifying Compilers

Compilers belong to a class of applications commonly referred to as language processors.[6]

A language processor can be described as a black box that takes structured input of a specific format A and transforms it into output of a specific format B. Conceptually speaking, the typical input for a user-facing compiler that produces an executable would be computer-encoded text, while its output would likely consist of machine code. Realistically speaking, language processors may also take any number of different inputs, produce any number of different outputs, or violate the functional abstraction[7] by having arbitrary side-effects.

It is expected that if a successful transformation between input and output is not possible, most probably because of erroneous input, the language processor instead provide an explanation of what went wrong during the transformation.[8]

Compilers are tasked with transforming input of a higher level of abstraction, meaning closer to a human-readable format into output of a lower level of abstraction, meaning closer to a machine-readable format. In contrast decompilers transform lower-level input into higher-level output, while transpilers are concerned with the transformation between formats of similar level.[9] Lastly optimizers produce output of the same format as their input, but improve this input with respect to one or more specific goals.[10]

The input of a compiler is often referred to as the source language or source while the output is called the target language or target. A format only used within the implementation of a language processor and thus not visible in its signature is also called an IR meaning intermediate representation.[11]

[6]Cf. Aho et al. 2007, pp. 1 sq.
[7]Cf. Hughes 1989, p. 2.
[8]Cf. Aho et al. 2007, p. 1.
[9]Cf. Kulkarni, Chavan, and Hardikar 2015, p. 1629.
[10]Cf. Lattner and Adve 2011, p. 156.
[11]Cf. Cooper and Torczon 2011, p. 6.

2.2 Composing Compilers

From the view of compilers as black boxes that transform between different formats, an obvious pattern emerges. Multiple language processors can be composed in a manner similar to that of pure mathematical functions.[12] As an example, a hypothetical transpiler with the signature $A \to B$ and a hypothetical compiler with the signature $B \to C$ could be combined producing a single unit with the signature $A \to C$.

This process is often significantly harder when one of the components has multiple associated inputs or outputs, or performs side effects. Thus such a design should be carefully considered or avoided entirely.[13]

The design of smaller units of language processors is highly advantageous because it allows for a high degree of encapsulation and in turn code reuse. Supposing multiple high-level formats A_n have to be transformed into a single low-level format C, duplicated work can be avoided by designing an additional format B which all of A_n can be translated into with relative ease. The supposedly more complex transformation from $B \to C$ then only needs to be implemented once. This more modular approach may also decrease the complexity of any singular part of the system and thus decrease cognitive burden.[14]

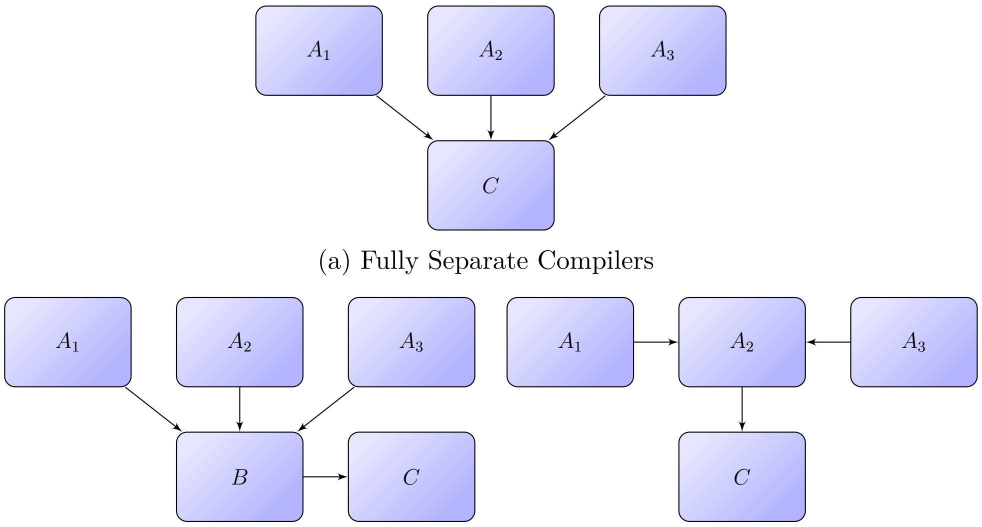
Alternatively this problem could also be solved using a transpiler from A_n to any specific high-level format, for example A_2, and then compiling only that specific high-level format into C. In practice, this third option is often taken when a suitable compiler for the high-level format, in this case $A_2 \to C$, already exists.[15]

Corresponding visualizations for the options introduced in the previous paragraphs may be found in Figure 1.

2.3 Compiler Components

While understanding the externally visible structure of a compiler is vital, it can only get us so far. In order to fully understand and successfully implement a compiler, we must also carefully consider its internal structure and components.

[12]Cf. Hughes 1989, pp. 9 sq.
[13]Cf. ibid., p. 9.
[14]Cf. Cooper and Torczon 2011, p. 223.
[15]Cf. Kulkarni, Chavan, and Hardikar 2015, p. 1630.



(a) Fully Separate Compilers

(b) Intermediate Representation

(c) Transpiler

Figure 1: Visualization of Common Ways to Compose Compilers (auth.)

In order to introduce these concepts, this section walks through a simplified version of a compiler transforming a subset of the WebAssembly textual format, hereafter referred to as Wat, to the WebAssembly binary format, hereafter referred to as Wasm.

I consider the Wat format suitable because, while it is intended for human consumption and exhibits many traits of high-level programming languages, it is also explicitly designed to be very close to, and easily transformable into, the lower-level Wasm binary format.[16] By using Wat/Wasm as examples, it is also hoped that these sections may serve a dual purpose in introducing the reader to some of their peculiarities, which, while not strictly necessary, might prove useful in later sections related to WebAssembly.

It should however be noted that the focus of this section does not lie on describing an efficient compiler realistically suitable to the task specified above. The process of compiling Wat to machine code presented here should only be viewed as an example used to illustrate basic concepts of compiler theory that are intended to be applicable to a wider range of tasks.

2.3.1 Lexical Structure

In its most rudimentary consideration, Wat is stored in the form of files containing computer encoded text, specifically using the UTF-8 encoding for Unicode characters.[17]

[16]Cf. Rossberg and W. C. Group 2019, p. 103.
[17]Cf. ibid., p. 103.

Storing Wat source code in such a textual format allows the programmer to easily read and edit this source code using common primitives of computer interaction, such as keyboard and screen, in order to create programs that can then be understood by a compiler.

Still, when writing even the simplest applications, programmers have to understand the code they produce on a level beyond that of a list of characters. As with natural languages certain sequences of or single special characters may compose larger meaningful units, called words in natural languages, that then again may be composed to form complete thoughts. In computer science this process of gradually improving the models by which to understand and tackle a problem is called abstraction.[18]

Much like human programmers, compilers also need to perform abstraction on the input they are given in order to reason about its meaning and provide meaningful output. One of the first steps many compilers take in abstracting away from a simple string of characters is converting this string into a sequence of meaningful units, commonly referred to as lexemes or tokens. This process is most often called lexical analysis or just lexing.[19]

Figure 2 shows a simple lexical analyzer or lexer, implemented in Rust pseudocode. Note that this lexer does not split its input into graphemes, which would be needed to reliably handle Unicode characters because of the variable-width nature of UTF-8.[20]

The lexing step thus serves to prepare the input for further transformation and analysis by removing unnecessary information such as whitespace or comments. It may also choose to reject certain input that is considered invalid regardless of context.[21]

It should be noted that the lexer described here, as a pure function from a string to a vector of tokens, again closely follows the structure described in Section 2.2, supporting the argument that a compiler itself is structured as a chain of other language processors. In this case, the input string could be considered the input format of the lexer while its target format would be a list of tokens. In the context of the overarching compiler, the list of tokens can then also be considered an intermediate representation.[22]

[18]Cf. Abelson and Sussman 1996, p. 35.
[19]Cf. Aho et al. 2007, pp. 5 sq.
[20]Cf. Bjarne 2013, p. 179.
[21]Cf. Cooper and Torczon 2011, p. 27.
[22]Cf. ibid., p. 222.

```rust
fn lex(input: String) -> Vec<Token> {
    let mut input = input.chars();
    let mut tokens = vec![];
    let mut buffer = String::new();

    // Read through all characters in the input string
    while let Some(char) = input.next() {
        let token = match char {
            '(' => Some(Token::OpenParen),
            ')' => Some(Token::CloseParen),
            // Potential other tokens...
            char => {
                if char.is_whitespace() {
                    None
                } else {
                    buffer.push(char);
                    continue;
                }
            }
        };

        // Handle special instructions
        match buffer.as_str() {
            "module" => tokens.push(Token::Module),
            "export" => tokens.push(Token::Export),
            "func" => tokens.push(Token::Function),
            // Potential other tokens...
            other => {
                tokens.push(Token::Identifier(other.to_string()));
                buffer = String::new();
            }
        };

        token.map(|ok| tokens.push(ok));
    }

    return tokens;
}

enum Token {
    OpenParen,
    CloseParen,
    Module,
    Export,
    Function,
    Identifier(String),
    // Potential other tokens...
}
```

Figure 2: Simple Lexical Analyzer Implemented in Rust Pseudocode (auth.)

2.3.2 Syntactic Structure

Using the method of lexical analysis described in the previous section allows us to understand our input at a level that more closely resembles the actual meaning of the source code. However, compilers for nearly all programming languages require further understanding of the structure of their input in order to correctly interpret it and perform meaningful work.[23]

To properly visualize this, consider Figure 3. Both code listings *a* and *b* would be considered valid by the pseudo-lexer implemented in Section 2.3.1, however only listing *b* also represents valid Wat code. Similarly, both the sentences *c* and *d* are composed of entirely valid English words, however only sentence *d* also expresses meaning.

```
)() module func ( export
```

(a) Syntactically Invalid Wat Code

```
(func $foo (result i32) i32.const 42)
```

(b) Syntactically Valid Wat Code

death death The. falls its whale

(c) Syntactically Invalid Sentence

The whale falls to its death.

(d) Syntactically Valid Sentence

Figure 3: Contrasting Lexical and Syntactic Correctness (auth.)

In order to verify the syntactic correctness and further analyze the input stream provided by the lexer, compilers use a technique called parsing. By parsing the input stream the parser component of the compiler performs a conversion from a one-dimensional data structure into a graph structure that also intends to capture the meaning of the original input. Figure 4 visualizes this process for a simple Wat program. This new data structure is generally called the abstract syntax tree or AST and is often used extensively during further analysis.[24]

The art of parsing one-dimensional input efficiently and correctly according to an arbitrary set of rules is an important area of research for both classical and computer linguistics. Compiler authors generally do not have to concern themselves with the details of these

[23]Cf. Cooper and Torczon 2011, p. 83.
[24]Cf. ibid., p. 227.

processes as there exist numerous tools able to automatically generate efficient parsers, or lexer-parser combinations, from comparatively simple descriptions.[25]

These descriptions, or more commonly grammars are often given in the form of context-free grammars or CFGs. The Backus-Naur form or BNF is one common way to describe such a grammar.[26] Context-free in this case means that any given definition or production of the grammar is defined only via the productions it contains.[27]

A grammar is thus context-free if no additional context, which means knowledge of other parts of the input stream not captured by the production currently being analyzed, is used. These restrictions allow generated parsers to parse most programming languages, if not natural languages,[28] in a determinate amount of time with regard to the length of the input and the complexity of the grammar.[29]

2.3.3 Semantic Structure

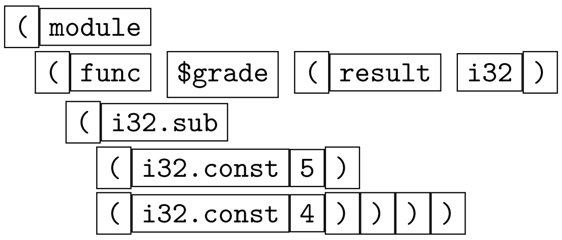
While the process of parsing our programming language ensures our input is syntactically correct, a context-free parser inherently cannot also ensure its semantic correctness.[30]

Consider Figure 5. Both *a* and *b* show Wat code that is syntactically correct, however the code *b* references an undefined global variable. Thus the code *b* could never execute correctly and has to be considered invalid.

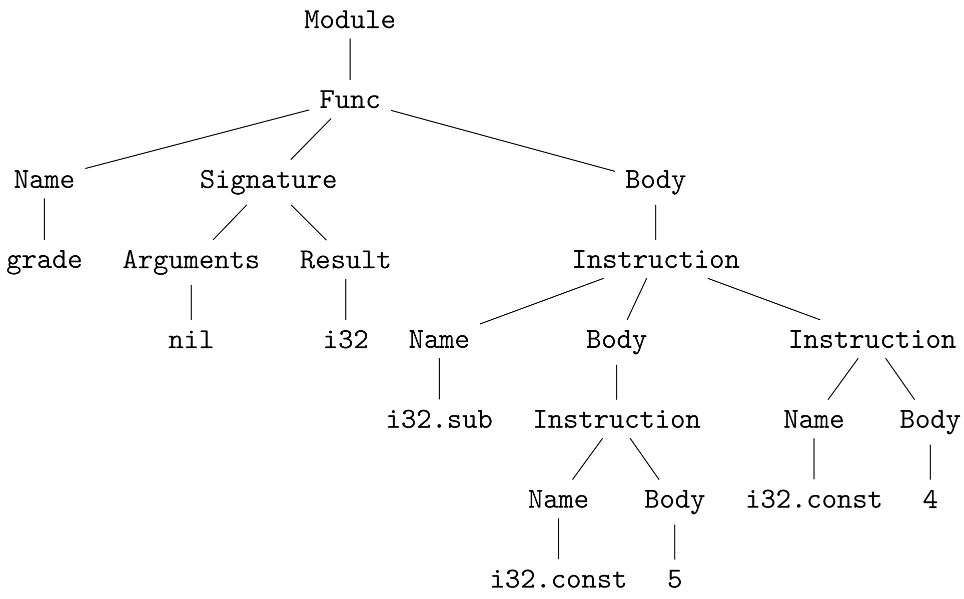
There are many kinds of semantic errors a snippet of code may exhibit. As an example, instead of the global variable in *b* not being defined, it could also be of an incorrect type or itself depend on other undefined variables.[31]

As the process of semantic analysis requires detailed information on all parts of the input, it often makes use of a data structure called the symbol table which stores information on all symbolic constructs, such as variables and functions, mapped to their names and corresponding scopes. The symbol table is most often implemented in terms of an auxiliary data structure, which is constructed during the lexing and parsing steps.[32]

[25]Cf. Aho et al. 2007, p. 37.
[26]Cf. Cooper and Torczon 2011, p. 87.
[27]Cf. ibid., pp. 111 sq.
[28]Cf. Aho et al. 2007, pp. 218 sq.
[29]Cf. Cooper and Torczon 2011, p. 109.
[30]Cf. ibid., p. 161.
[31]Cf. Rossberg and W. C. Group 2019, pp. 19 sqq.
[32]Cf. Aho et al. 2007, pp. 86 sq.



(a) Input Given to the Parser



(b) Possible Output of the Parser

Figure 4: Visualization of the Syntactic Analysis Phase (auth.)

```
(module
 (global $var i32 (i32.const 10))
 (func $start (result i32)
  (global.get $var)))
```

(a) Semantically Correct Wat Code

```
(module
 (func $start (result i32)
  (global.get $var)))
```

(b) Semantically Incorrect Wat Code

Figure 5: Contrasting Syntactic and Semantic Correctness (auth.)

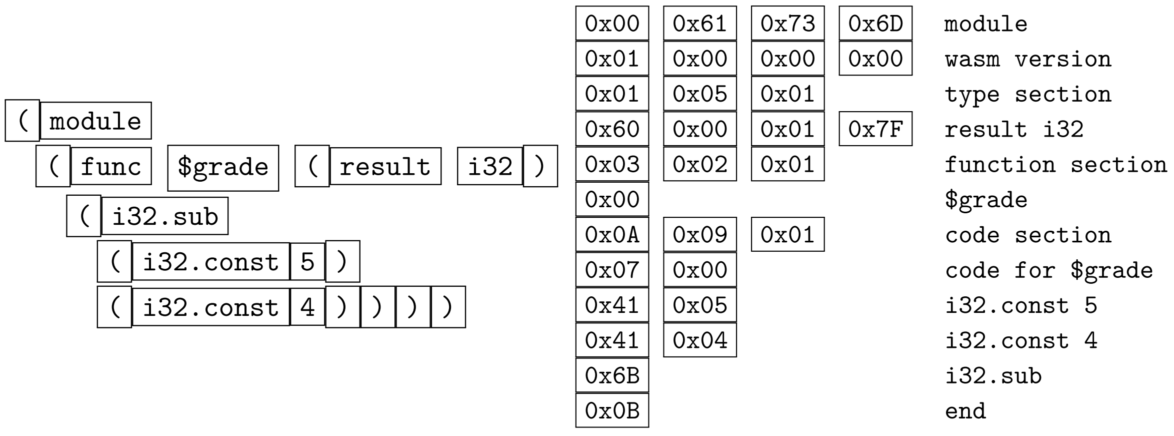
A typical semantic analysis procedure simply walks through the abstract syntax tree and recursively checks for the semantic correctness of every node in accordance with both the symbol table and other nodes in the abstract syntax tree.[33]

During the process of semantic analysis, both the abstract syntax tree and symbol table may also be mutated as additional information gathered during syntactic analysis can be used to simplify further steps.

2.3.4 Code Generation

Lastly, a compiler needs to generate some output of the target format. This process is generally called code generation.

In our current model, as the previous steps have already resulted in an abstract syntax tree that is known to be semantically correct and closely follows the structure of the target binary Wasm, the code generation procedure simply has to recursively walk the abstract syntax tree and emit the correct binary code fragments for every node it encounters, as defined by the WebAssembly specification.[34] As shown in Figure 6 this constitutes a transformation from a tree to a linear one-address representation.[35]



```
(module
  (func $grade (result i32)
    (i32.sub
      (i32.const 5)
      (i32.const 4))))
```

```
0x00 0x61 0x73 0x6D   module
0x01 0x00 0x00 0x00   wasm version
0x01 0x05 0x01        type section
0x60 0x00 0x01 0x7F   result i32
0x03 0x02 0x01        function section
0x00                  $grade
0x0A 0x09 0x01        code section
0x07 0x00             code for $grade
0x41 0x05             i32.const 5
0x41 0x04             i32.const 4
0x6B                  i32.sub
0x0B                  end
```

(a) Wat Input Used for Code Generation (b) Possible Wasm Output

Figure 6: Visualization of the Code Generation Phase (auth.)

More generally, the exact nature of the code generation step derives from the relative level of abstraction between its source and target. As such, conceptually incompatible properties of the source and target languages may substantially complicate code generation.[36]

[33]Cf. Aho et al. 2007, p. 8.
[34]Cf. Rossberg and W. C. Group 2019, pp. 87 sqq.
[35]Cf. Cooper and Torczon 2011, pp. 235 sq.
[36]Cf. ibid., p. 242.

2.4 Further Steps

The previous sections have been concerned with the implementation of a compiler for a programming language. However, it could be argued that a complete programming language is equally or more so defined by the functionality it provides to its user.

While some of this functionality is embedded into the language through certain language features, it is also common for high-level programming languages to provide a selection of reusable routines implemented, or partially implemented, using the language itself.[37] This so-called standard library may then be used by users of the language in order to fulfill common tasks that would otherwise have to be implemented by them.[38]

This approach is advantageous because it allows the programmer to treat the standard library in the same manner as any other code written by them, potentially improving modularity. Furthermore, the existing compiler and any associated tooling for working with the language could also be used to further improve the standard library.[39]

[37]Cf. Bjarne 2013, p. 38.
[38]Cf. ibid., p. 860.
[39]Cf. ibid., p. 17.

3 Designing “wrk”

While the main focus of this paper lies in implementing a compiler for a new language, first designing this language is a prerequisite to this task. The design of the language, dubbed “wrk”, has evolved with the creation of both this paper and the reference compiler.

3.1 Design Goals

In order to restrict the scope of the “wrk” language, some goals on what the language and its eventual compiler should achieve were set.

Firstly, the language should exhibit features that facilitate the creation of a simple but effective application that could be presented to a non-technical audience in a presentation context. This choice was made because material on preparing successful presentations, provided by the Austrian Ministry of Education, Science and Research, suggest that students should try to distinguish their presentation by incorporating their work and results.[40] Compiling and executing a working application in front of the committee would achieve exactly this.

Secondly, the “wrk” language and compiler should exhibit at least one idea the author personally considers novel. As Perils famously states, *“A language that doesn’t affect the way you think about programming, is not worth knowing.”*[41] I agree with this assessment and thus would like to ensure that the “wrk” language exhibits at least one trait or combination of traits that distinguishes it from already existing programming languages. It is also clear that, while any such concept might be novel to me, it will likely already have been examined and understood in prior work which I am simply unaware of.

3.2 Choosing a Compilation Target

As per the design introduced in Section 2, any compiler or rather compiler toolchain for a language of a higher abstraction level necessarily consists of at least one front- and one backend component. These components then traditionally interface through the means of a well-defined intermediate representation.[42]

[40]Cf. Tscherne et al. 2019.
[41]Perlis 1982, p. 8.
[42]Cf. Lattner and Adve 2011, pp. 155 sq.

As the strength of any compilation target is heavily dependent on the strength of its associated ecosystem and, perhaps even more importantly, because implementing a completely new backend represents potentially huge implementation cost, opting for a predefined, sufficiently high-level compilation target seems reasonable.[43]

Choosing such a target however may also be considered a challenging task in and of itself, as any particular option could majorly influence the capabilities and by extension also the design of the source language that it is targeted by.[44]

During the early stages of the "wrk" project two potential targets, LLVM IR and WebAssembly, were considered. The following sections consist of informal descriptions of these targets and an analysis of their respective merits and drawbacks.

3.2.1 LLVM

The LLVM project can be described as a collection of intercompatible tools suitable for building a low-level compiler toolchain.[45] Many of these components act on the so-called LLVM IR, a low-level intermediate representation also specified by the LLVM project. Compilers that intend to use LLVM as part of their toolchain are generally expected to target LLVM IR.[46]

Most of the tools provided by LLVM are in large parts implemented as C++ libraries facilitating uncomplicated integration into C++ applications.[47] Official C bindings are also provided, while other languages generally require the usage of either manual foreign function interfaces or third-party libraries to effectively utilize LLVM components.[48]

At the time of my research into potential compilation targets no high-level libraries exposing LLVM functionality to the implementation language of the compiler, Rust, existed. Since then, this has changed and multiple such libraries are now available.

[43]Cf. Lattner and Adve 2011, p. 158.
[44]Cf. Cooper and Torczon 2011, p. 264.
[45]Cf. Lattner and Adve 2011, p. 155.
[46]Cf. ibid., pp. 157 sq.
[47]Cf. ibid., pp. 160 sq.
[48]Cf. ibid., p. 166.

3.2.2 WebAssembly

WebAssembly is a comparatively new technology standardized by the WebAssembly Working Group with major support from browser vendors, such as Mozilla, Google, Microsoft and Apple.[49] While it has been developed mainly in order to provide a fast and safe extension to the existing methods for code execution present in modern browsers such as JavaScript, the design of WebAssembly also allows for its use in areas unrelated to the internet.[50] Thus, using WebAssembly, it is theoretically possible to create applications that may run as part of websites accessible on the net, as well as natively executed desktop or mobile applications.

WebAssembly is described as a low-level code format optimized for compact representation and safe, efficient execution. Notably, WebAssembly programs are expected to be sandboxed and embedded in a host environment or runtime and may only make use of additional procedures that have explicitly been provided by the environment, such as JavaScript functions in case of a web embedding, to operate outside of their sandbox.[51] More generally WebAssembly applications may import or share numeric values, functions, function tables and regions of linear memory which are one-dimensional memory arrays potentially suitable for storing larger amounts of data, from or with their environment.[52]

The WebAssembly Community Group also defines the WebAssembly textual format, a format intended to display WebAssembly code as human-readable text.[53] Tools for validating, converting between the different formats of, and generally operating on WebAssembly are provided by the WebAssembly Community Group, while further tooling and WebAssembly runtimes are developed by external providers.

As both WebAssembly and Rust are majorly backed by the Mozilla Foundation, much of this tooling is in fact implemented in Rust. This facilitates relatively straightforward interaction with WebAssembly using native and idiomatic Rust code.

[49]Cf. Watt 2018, p. 53.
[50]Cf. Rossberg and W. C. Group 2019, p. 1.
[51]Cf. Watt 2018, p. 53.
[52]Cf. Rossberg and W. C. Group 2019, p. 44.
[53]Cf. ibid., p. 103.

3.2.3 Result

After evaluating both LLVM and WebAssemby, the latter was chosen as the most suitable target for the “wrk” compiler for a combination of the reasons outlined below.

WebAssembly, by virtue of having a smaller specification was found to be much simpler to understand than LLVM IR. Additionally the quality of entry-level documentation for WebAssembly was found to be much higher than that of its LLVM counterparts.

In addition to that, the Rust ecosystem contains many tools and libraries that simplify working with WebAssembly, including tools for code generation, validation and several runtimes. Equivalent facilities for working with LLVM seemingly either operate on a much lower level of abstraction, or are simply nonexistent.

Lastly, the fact that a WebAssembly application can easily be embedded inside a web page much simplifies the potential act of showcasing a running application in a presentation context. Existing web technologies could then be used to better illustrate important characteristics of the application to a non-technical audience.

3.3 Informal Definition

In the state which is described in this paper, “wrk” constitutes a simple, statically typed, functionally inspired language with a syntax and type system superficially similar to that of Haskell.[54] While its semantics are largely functional, the language provides first-class support low-level access to memory. It also has to be noted that many integral concepts important to a fully-featured functional programming language, such as proper closures and custom data types, have been left out of the current iteration of the design because of time constraints.

The following sections represent an informal and incomplete specification of the “wrk” language in its current state.

3.3.1 Syntax

The syntax of the “wrk” language is approximately described by the BNF grammar shown in Figure 7. For the sake of brevity, this description has deliberately been kept undecidable

[54]Cf. Marlow 2010, p. 135.

in places where, for example, precedence relationships would unnecessarily complicate the grammar and certain obvious nonterminals have been left out.

At the file level, there exist two possible entry points into the grammar. `library`, which simply represents a list of variable and function definitions and `application`, which additionally defines a list of entry points identified by the `"export"` identifier.

A potential compiler is also expected to filter out lines starting with `"--"` as comments, which should not affect the meaning of the source code in any way.

3.3.2 Evaluation Semantics

Evaluation of functions and values is performed lazily in accordance with "wrk" evaluation semantics. This in practice means that these expressions are evaluated only when used as part of other expressions that also have to be evaluated.[55]

Symbols specified as entry points into a "wrk" application represent an exception to this rule as they are exported in the form of WebAssembly functions, which may then be strictly evaluated. This, in combination with built-in instructions that may perform side-effects, allows for effectful programming using the "wrk" language.

These informal semantics provide no special guarantees relating to program failure or termination when working with or without side effects.

3.3.3 Type System

The "wrk" language features a static type system with support for generic types and type inference. In effect, this allows for the static validation of "wrk" programs by the compiler without forcing the programmer to explicitly annotate expressions with types.[56]

The syntax for types used in the "wrk" language, as well as this paper, is heavily inspired by that of the Haskell language.[57] An index of supported types is given in Figure 8. Lowercase and uppercase letters, such as `a` and `M`, are shorthands used in this paper to describe generic types and generic memory region types. They are not a valid part of the "wrk" type syntax.

[55]Cf. Hughes 1989, p. 9.
[56]Cf. Aho et al. 2007, p. 387.
[57]Cf. Marlow 2010, pp. 37 sq.

The type system also shows explicit support for what the “wrk” language calls memory regions or memories. Memory regions differ from other value types as they generally only exist in the form of references to other memory regions. As such, parts of memory regions are passed to functions per reference. Functions may also automatically return regions of memory given to them as an argument, most probably after they have been mutated by this respective function. This has to be explicitly specified using the special `&` symbol as their return type. This system aims to facilitate intuitive as well as safe access to memory and potentially shared memory.

3.3.4 Built-In Instructions

The “wrk” language defines a set of built-in instructions that may be used in the same manner as user-defined functions and values.

Built-in instructions also serve a special role with regard to the type system of the language, as they, unlike user-defined functions, which are either fully generic or accept only concrete types, may qualify on certain sets of types. This, for example, allows for the implementation of certain generic mathematical operators such as `!add`, which may be used to add both two integers as well as two floating-point numbers.

As can be seen in Figure 9, which shows a listing of some of the instructions the “wrk” language defines, some of these instructions, such as `!not`, `!and`, `!add`, or `!sub`, indeed constitute low-level logical or mathematical operations, while others, such as `!request`, allow for access to memory regions. Others again, here `!display`, provide much higher-level functionality mainly useful for quickly demonstrating certain language features or features of an eventual compiler.

```
application ::= definitions "export" symbols
library ::= definitions

definitions ::= definition ";" | definition ";" definitions
definition ::= symbol "=" expression
             | symbol "::" type "=" expression

expression ::= "if" expression "then" expression "else" expression
             | "let" definitions "in" expression
             | symbol ":" expression
             | expression value
             | value "::" type
             | value

value ::= "(" expression ")"
        | bool
        | integer
        | floating
        | string
        | builtin
        | symbol

type ::= type "->" type
       | memory "->" "&"
       | "[" memory "]"
       | "Int"
       | "Float"
       | "Bool"
       | "*"

memory ::= ".."
         | index ".."
         | sized_memory "#" memory

sized_memory ::= ".." index
               | index ".." index
               | sized_memory "#" sized_memory

index ::= unsigned_integer
        | index "+" index
        | index "*" index

string ::= "\"" characters "\""
builtin ::= "!" symbol
bool ::= "true" | "false"

symbols ::= symbol | symbol "," symbols
symbol ::= lowercase_character | lowercase_character characters
```

Figure 7: BNF Description of the “wrk” Language Syntax, Simplified (auth.)

Representation	Example	Description
`a, b, c, ..`	`a`	Any type, shorthand
`A, B, C, ..`	`M`	Any memory region type, shorthand
`*`	`*`	Any type
`Bool`	`Bool`	Boolean value
`Int`	`Int`	Integer number, 32-bit
`Float`	`Float`	Floating-point number, 64-bit
`a -> b`	`Int -> Float`	Function from type a to b
`M -> &`	`[..10] -> &`	Function from type M to M, implicitly returning M
`[..]`	`[..]`	Unsized memory region
`[i..]`	`[3..]`	Unsized memory region starting at offset i
`[..i]`	`[..10]`	Memory region spanning offset 0 to i
`[..i+j]`	`[..10+3]`	Memory region spanning offset 0 to $(i + j)$
`[..i*j]`	`[..10*3]`	Memory region spanning offset 0 to $(i * j)$
`[i..j]`	`[3..10]`	Memory region spanning offset i to j
`[i..j#k..l]`	`[3..5#7..10]`	Memory region spanning offset i to j and k to l

Figure 8: Index of "wrk" Type Syntax, Simplified (auth.)

Name	Typing	Description
`!eq`	$a \mapsto a \mapsto Bool$	Test whether the given arguments are equal
`!not`	$Bool \mapsto Bool$	Logical negation on the given Boolean
`!and`	$Bool \mapsto Bool \mapsto Bool$	Logical conjunction on the given Booleans
`!or`	——〃——	Logical disjunction on the given Booleans
`!xor`	——〃——	Logical exclusive or on the given Booleans
`!add`	$a \mapsto a \mapsto a \mid a \in \{Int, Float\}$	Add given numbers
`!sub`	——〃——	Subtract second from first given number
`!mul`	——〃——	Multiply given numbers
`!div`	——〃——	Divide first with second given number
`!request`	$M \mid M$ is sized	Request a region of zeroed memory fitting M
`!slice`	$[..i] \mapsto Int \mapsto [..j]$	Restrict given memory region to given max
`!take`	$[..i * j] \mapsto Int \mapsto [..j]$	Restrict given memory region to given row
`!display`	$M \mapsto M \mid M$ is sized	Display given memory region as a pixel grid

Figure 9: Listing of Some Built-In Instructions of the "wrk" Language (auth.)

4 Implementing “wrk”

Over the course of 2019 and 2020 a prototype implementation of a compiler for the “wrk” language was developed as part of this paper.

As suggested in Section 3.2 the implementation explicitly targets WebAssembly and supports generating code suitable for both web and non-web embeddings. The compiler also supports emitting basic HTML and JavaScript boilerplate, which facilitates demonstrations of “wrk” code running in the browser, and makes use of the Wasmer WebAssembly runtime in order to directly execute generated code in a non-web context.

4.1 Project Structure

As the prototype compiler is implemented in Rust, we make extensive use of its module system[58] for organizing our code. Figure 10 shows the approximate module and file structure used by the compiler.

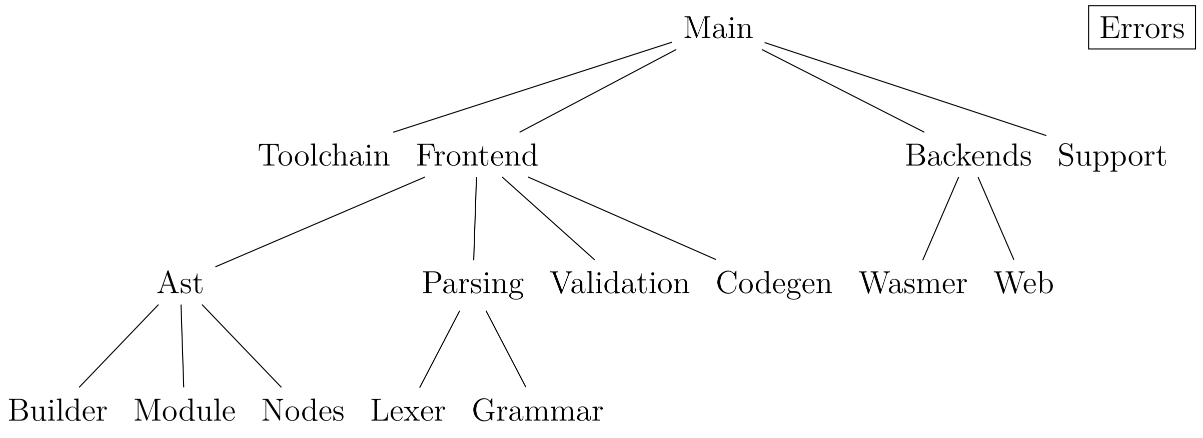


Figure 10: Approximate Module Structure Used by the Compiler Implementation (auth.)

Having access to an expressive module structure was extremely helpful in continuously refactoring the codebase as it evolved with the design of the language and the structure of the overall project.

In the final iteration of the project, most interesting parts of the compiler live in the `frontend` module. The modules `parsing`, `validation` and `codegen` respectively are concerned with the tasks of parsing, validating and generating WebAssembly output from

[58]Cf. Klabnik and Nichols 2018, pp. 109 sqq.

“wrk” code. The `backends` module concerns itself with presenting “wrk” applications using one of the supported backends, namely the Wasmer WebAssembly runtime and a web page embedding suitable for execution in a web browser that supports WebAssembly. Several other submodules that provide more generic library functionality live in the `support` module. Most modules also contain `errors` submodules that define suitable errors for the failures they may exhibit.

The functionality of the compiler `frontend` and `backend` is chained together by the `toolchain` module, which also provides rudimentary logging and interactive progress reporting. Lastly, the `main` module simply provides a command-line interface for invoking the functionality provided by the `toolchain` module.

4.2 Interesting Implementation Details

As one of the main foci of this paper was implementing a working compiler for the “wrk” language using the principles of modern programming language and compiler research, some liberties were taken in implementing core parts of the prototype compiler. This in turn has resulted in several components and data structures used heavily deviating from the norms described in classical introductions to compiler design.

I feel it is important to examine these choices, analyze their merits and drawbacks, as well as possibly evaluate the viability of their use in larger, more professional compiler projects.

4.2.1 Arena-Based AST

The reference compiler implementation uses a tree implemented through the custom data type `Arena` as the primary structure for storing the abstract syntax tree representation of a “wrk” program.

Internally, a growable array or vector[59] is used to store items of a single data type. By associating these data types with indices into this same environment, a conceptual tree structure, such as the one seen in Figure 11 may be formed. However, it should be understood that this tree structure is not paramount and any data type may in fact be stored inside the `Arena`.

```rust
use crate::support::env::{Identifier, Arena};

type Tree = Arena<Node>;

#[derive(Default)]
enum Node {
    Branch(Identifier, Identifier)
    Leaf,
}
```

Figure 11: Simple Binary Tree Structure Implemented through `Arena` (auth.)

This particular structure was chosen for a number of reasons. Most importantly it solves the problem of associating arbitrary information with the AST,[60] which was encountered

[59] Cf. R. D. W. Group 2019.
[60] Cf. Yang 2013.

very early into the implementation process. In order to associate any information with the AST we may simply define a new Arena that stores information using the same identifier values as the Arena representing the AST.

Consider the simplified code snippet in Figure 12, which is used as part of the compiler's type inference algorithm. It ensures that AST nodes do not have to go through the expensive type inference procedure multiple times. To achieve this, we define an additional Arena of Boolean values that indicate whether a particular node has already been visited and update it accordingly.

```
fn infer(
    id: Identifier,
    check: &mut Arena<bool>,
    // Other...
) -> Result<Type, Error> {

    // Setup code...

    let visited = check.get_mut(id);
    if *visited {
        // Short-circuit algorithm
        return Ok(tp.clone())
    } else {
        // Change visited status to true
        *visited = true;
    }

    // Expensive inference algorithm...
}
```

Figure 12: Type Inference Procedure Showing an Use-Case for Arena, Simplified (auth.)

It should also be expected that this structure is relatively time and space efficient for the tasks of building, storing and walking the AST. By approximating the number of expected nodes, code using Arena may pre-allocate sufficient space so no further allocations are required. Indices are used as-is to index into the internal array, so the Arena has best-case efficiency of $O(n)$ with regard to the number of stored nodes. However, the worst-case space efficiency for code that does not insert values into the array with approximately consecutive identifiers is only bound by the maximum possible value of an identifier. Node lookup and insertion only requires one bound-checked array indexation, which is $O(1)$ with regard to the size of the internal array and also generally expected to be cheap.[61]

[61] Cf. R. D. W. Group 2019.

Still, this approach also comes with its own set of problems. Code walking the flat AST needs access to the full `Arena` at all times, essentially mandating that code that needs to mutate the AST may not be parallelized in the same way as a code mutating a typical tree structure. Code mutating arena-based trees may also be somewhat more verbose than comparable code using traditional boxed trees. Furthermore, the very general design of the `Arena` allows for AST implementations that are relatively unprincipled. In turn the AST used in the reference compiler implementation does not necessarily encode the validity of the tree in its data type, requiring dependent code to simply assume it has been assembled in the correct manner.

4.2.2 Combined AST and Symbol Table

As mentioned in the previous section, `Arena` allows storing data that follows not just a regular tree pattern, but any pattern. We can make use of this property by defining an AST that stores not only the AST nodes, but also understands their symbolic relationship using absolute identifiers into the `Arena`.

```rust
pub use crate::support::env::Identifier;

#[derive(Clone, Debug, PartialEq)]
pub enum ExprKind {
    Lambda(Identifier, Identifier),
    App(Identifier, Identifier),
    Integer(i32),
    Floating(u64),
    Boolean(bool),
    Symbol(Identifier),
    If(Identifier, Identifier, Identifier),
    Let(Vec<Identifier>, Identifier),
    Is(Identifier),
    Def(Identifier),
}
```

Figure 13: AST Implementation Used in the Reference Compiler, Simplified (auth.)

As seen in Figure 13, the `Symbol` variant of the `ExprKind` enumeration is used to identify that a particular AST node represents a symbol. Other procedures related to validation or code generation may then use this information to resolve these symbols without needing any access to an external symbol table.

This particular approach is useful because it allows us to see the AST as the only source of information about for further analysis. Thus no symbol lookup or resolution has to be performed after the AST has been assembled. However it also prevents us from treating our AST as a rooted tree or forest. In fact our AST no longer follows a tree pattern and is instead best described as an directed acyclic graph or DAG.[62] This fact has not fully been taken advantage of in the present implementation of the reference compiler. It seems likely that some operations on the AST could be simplified by properly reconsidering its properties and role as a DAG.

4.2.3 Error Discoverability Using the Rust Type System

As Czaplicki suggests, good error messages are vital to the experience of using any compiler.[63] The reference compiler makes use of the Rust type system and derive features to facilitate frictionless definition and usage of errors in its codebase.

Figure 14 shows a typed error definition from the reference compiler codebase. As shown in Figure 15, this error also carries typed contextual information and may additionally be augmented with location information.

One advantage of this approach is that existing Rust tooling, namely RustDoc, could be used to automatically generate helpful human-readable documentation from these descriptions.[64] Such automatically generated documentation could help users of the compiler to quickly diagnose issues with their code.

[62]Cf. Aho et al. 2007, pp. 358 sq.
[63]Cf. Czaplicki 2015.
[64]Cf. Klabnik and Nichols 2018, p. 287.

```rust
use crate::support::codeerror::{Diagnostic, DiagnosticAt};
use displaydoc::Display;

#[derive(Clone, Debug, Display, Diagnostic, PartialEq)]
pub enum Error {
    /// Multiple definitions for the same name
    #[diag(short = "Name redefined here")]
    MultipleDefinitions {
        #[diag(label)]
        original: DiagnosticAt<Label>,
    },
    /// Name missing from containing scopes
    #[diag(short = "This name")]
    MissingSymbol,

    // Potential other errors...
}

#[derive(Clone, Debug, Display, Diagnostic, PartialEq)]
pub enum Label {
    /// Maybe remove this conflicting definiton
    #[diag(short = "Original definition")]
    OriginalDefinition,

    // Potential other labels...
}
```

Figure 14: Error Definition Used in the Reference Compiler, Simplified (auth.)

```rust
use crate::support::codeerror::{Diagnostic, Diagnostics};
use errors::{Error, Label}

fn return_example_error() -> Result<(), Diagnostics> {
    let label = Label::OriginalDefinition.at(0..2);
    let error = Error::MultipleDefinitions {
        original: label,
    };

    Err(error.at(10..12))?
}
```

Figure 15: Exemplary Function That Returns an Error with Contextual Information (auth.)

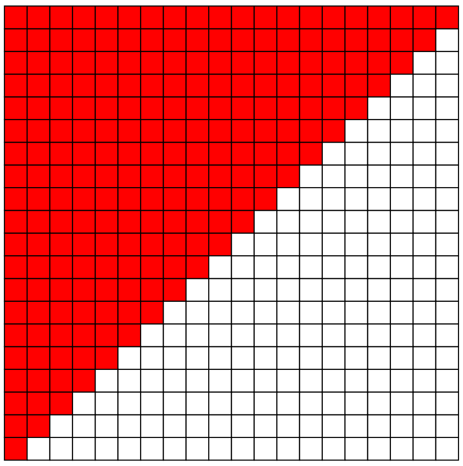
5 Evaluating the Success of the Project

The previous sections have discussed both the design and implementation of the “wrk” language. To conclude this project, possible applications of the resulting language and compiler are shown by examining an exemplary application implemented in the “wrk” language. Merits and drawbacks of the language and compiler in their present state are also discussed and an outlook on future developments is given.

5.1 An Example Application

```
run = let
  mem = !request :: [..20*20*4];
in !display (update 20 mem);

update = n: m:
  if (!eq n 0)
  then m
  else update (!sub n 1) (draw m n);

draw :: Int -> [..20*20*4] -> &
  = n: m: let
      row = (!sub 20 n);
    in draw_line n (!take m row);

draw_line :: Int -> [..20*4] -> &
  = i: m: !write_all color (!slice m i);

-- RGBA #FF0000FF
color = 4278190335;

export run
```

(a) Implementation



(b) Output

Figure 16: Exemplary “wrk” Application and Its Output (auth.)

Figure 16 *a* shows the implementation of a “wrk” application which, when compiled and executed by calling the exported `run` function, should produce and display a pixel grid as shown in *b*.

The application presents most language features currently present. We first request a region of memory from the WebAssembly runtime and then continuously update it row per row before displaying it as a pixel grid.

The `update` function uses recursion to repeatedly call `draw` and `draw_line` which then restrict the region of memory which is eventually written to using the `!write_all` built-in. Also consider how the `draw` function makes use of the special $M \mapsto \&$ type to return the memory region given to it as an argument.

The RGBA color value is represented by a 32-bit integer value which exactly matches the required bit width for 4 8-bit color values.

We can also see that, while the `draw` and `draw_line` functions need to be explicitly typed to handle their memory region arguments, other functions and values may largely infer their typing.

5.2 Language Ergonomics

As can be seen in the application presented in Section 5.1, features of the "wrk" language in their current form facilitate working with primitive values, such as integers, floating-point numbers and regions of memory, in the context of functional programming. Higher-level features such as user-defined types and proper closures are left out of the specification. The language also lacks a system for structuring and importing code into and from modules or libraries. As such, building larger and more complex applications using "wrk" would likely be unfeasible.

Support for working with memory regions can be considered the most unique feature of the "wrk" language. At its core lies the restriction to access of memory regions through static types that define their boundaries. Special built-in instructions, such as `!take` or `!slice`, may then be used to further restrict this access to smaller regions that can meaningfully be mutated using other built-in instructions, such as `!write_all`. This system is further helped by the special $M \mapsto \&$ function type which simplifies the common task of mutating a restricted region of memory.

Still, it remains to be seen if such a system would be viable or even useful for implementing applications that serve more than a simple demonstration purpose.

5.3 Compiler Usage

Apart from the theoretical design of the language, its implementation in form of the reference compiler is also very relevant to evaluating the overall project.

The “wrk” reference compiler helps users by providing quality error messages that aim to succinctly explain the problem at their cause, as well as give suggestions on possible solutions. Figure 17 shows one such error message, as outputted by the compiler when executed from a computer terminal that supports colored text.

```
error: Multiple definitions for the same name

  ┌── ./mod.wrk:4:1 ──
  │
4 │ foo = 2;
  │ ^^^ Name redefined here
  ·
2 │ foo = x: y: x;
  │ --- Original definition
  │
  = Maybe remove this conflicting definiton
```

Figure 17: Exemplary Error Message Given by the “wrk” Compiler (auth.)

It can be said that the error messages provided by the compiler already provide a suitable general idea about problems with any particular piece of “wrk” code. However many error messages, in particular those related to type and memory type errors, would most certainly profit from more specific descriptions and suggestions. The reference compiler also does not currently emit any warning messages, which means information about potential issues that do not inhibit compilation.

The time needed for compilation and the quality of the produced code also serve as important metrics for the viability of the compiler when working on more complex projects. However, without any such projects, no investigations of this kind have been made and further research would have to be done to properly address these considerations.

Lastly, the correctness of the reference compiler would have to be assessed. As large parts of the compiler codebase remain untested at this time, it is almost certain that not all possible edge cases are handled as expected.

5.4 Future Developments

In their current state, the “wrk” language and compiler provide very little utility for writing complex and useful programs. Important concept remain either unspecified or simply unimplemented. However, it is my personal opinion that, assuming further development of the project, it could provide real benefits in certain niche scenarios, where available memory is limited or needs to be manipulated very precisely.

Further developments would likely include, but not be limited to, support for modules and imports, user-defined types, a configurable memory allocator and a comprehensive standard library, which makes use of these and already existing features. Existing core features of the language, such as support for typed memory regions, would most probably also have to be adapted to support these potential higher-level abstractions. Lastly, this extended specification of the “wrk” language would have to fully be formalized.

However, it is increasingly unlikely for any such developments to be realized, as I personally consider the “wrk” project complete in the state which is described in this paper. My future efforts in the direction of language design, if any, would likely focus on a completely new design and implementation, possibly inspired by the findings of this paper.

References

[1] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs - 2nd Edition.* In collab. with Julie Sussman. 2nd ed. Cambridge, MA, USA: The MIT Press, 1996. ISBN: 978-0-262-01153-2.

[2] Alfred V. Aho et al. *Compilers: Principles, Techniques, & Tools.* 2nd ed. Boston, MA, USA: Pearson/Addison Wesley, 2007. ISBN: 978-0-321-48681-3.

[3] Stroustrup Bjarne. *The C++ Programming Language, 4th Edition.* 4th ed. Boston, MA, USA: Addison Wesley, 2013. ISBN: 978-0-321-56384-2.

[4] Keith Cooper and Linda Torczon. *Engineering a Compiler.* 2nd ed. Burlington, MA, USA: Morgan Kaufmann, 2011. ISBN: 978-0-12-088478-0.

[5] Evan Czaplicki. *Compiler Errors for Humans.* June 30, 2015. URL: `https://elm-lang.org/news/compiler-errors-for-humans` (visited on 01/13/2020).

[6] Rust Documentation Working Group. *Std::Vec::Vec - Rust.* 2019. URL: `https://doc.rust-lang.org/std/vec/struct.Vec.html` (visited on 01/01/2020).

[7] John Hughes. "Why Functional Programming Matters". In: *The Computer Journal* 32.2 (Feb. 1, 1989), pp. 98–107. ISSN: 0010-4620, 1460-2067. DOI: `10.1093/comjnl/32.2.98`.

[8] Steve Klabnik and Carol Nichols. *The Rust Programming Language.* San Francisco, CA, USA: No Starch Press, 2018. ISBN: 978-1-59327-828-1.

[9] Rohit Kulkarni, Aditi Chavan, and Abhinav Hardikar. "Transpiler and It's Advantages". In: *International Journal of Computer Science and Information Technologies* 6.2 (2015), pp. 1629–1631. ISSN: 0975-9646.

[10] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *The Architecture Of Open Source Applications.* Ed. by Amy Brown and Greg Wilson. The Architecture Of Open Source Applications 1. Raleigh, NC, USA: lulu.com, 2011, pp. 155–170. ISBN: 978-1-257-63801-7.

[11] Simon Marlow, ed. *Haskell 2010 Language Report.* 2010. URL: `https://www.haskell.org/definition/haskell2010.pdf` (visited on 03/07/2019).

[12] Alan J. Perlis. "Special Feature: Epigrams on Programming". In: *SIGPLAN Not.* 17.9 (Sept. 1982), pp. 7–13. ISSN: 0362-1340. DOI: `10.1145/947955.1083808`.

[13] Andreas Rossberg and WebAssembly Community Group. *WebAssembly Specification.* 2019. URL: `https://webassembly.github.io/spec/core/_download/WebAssembly.pdf` (visited on 09/22/2019).

[14] Karin Tscherne et al. *Präsentation und Diskussion vorbereiten.* Sept. 2019. URL: `https://www.ahs-vwa.at/lehrpersonen/betreuungsprozess/praesentation-und-diskussion-vorbereiten` (visited on 01/03/2020).

[15] Conrad Watt. "Mechanising and Verifying the WebAssembly Specification". In: *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Los Angeles, CA, USA). CPP 2018. New York, NY, USA: Association for Computing Machinery, 2018, pp. 53–65. ISBN: 978-1-4503-5586-5. DOI: `10.1145/3167082`.

[16] Edward Z. Yang. *The AST Typing Problem.* May 23, 2013. URL: `http://blog.ezyang.com/2013/05/the-ast-typing-problem/` (visited on 01/01/2020).

List of Figures

Statement of Authorship

I hereby declare that I completed this paper on my own and that information which has been directly or indirectly taken from other sources has been noted as such. Neither this nor a similar work has been presented to an examination committee.

The template for this Statement of Authorship has helpfully been provided by Robin Neumann for use under the terms of the MIT License. It has been adapted by the author for use in this paper.