COMPILATION OF A TYPED R-LIKE LANGUAGE TO WEBASSEMBLY

Baljinnyam Bilguudei

Bachelor's thesis
Faculty of Information Technology
Czech Technical University in Prague
Department of Computer Science
Study program: Informatics
Specialisation: Computer Science
Supervisor: Pierre Donat-Bouillud, Ph.D.
December 29, 2025

Citation of this thesis: Baljinnyam Bilguudei. *Compilation of a typed R-like language to WebAssembly* . Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2026.

Chtěl bych poděkovat především sit amet, consectetuer adipiscing elit. Curabitur sagittis hendrerit ante. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Cras pede libero, dapibus nec, pretium sit amet, tempor quis. Sed vel lectus. Donec odio tempus molestie, porttitor ut, iaculis quis, sem. Suspendisse sagittis ultrices augue.

Declaration

FILL IN ACCORDING TO THE INSTRUCTIONS. VYPLŇTE V SOULADU S POKYNY. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Curabitur sagittis hendrerit ante. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Cras pede libero, dapibus nec, pretium sit amet, tempor quis. Sed vel lectus. Donec odio tempus molestie, porttitor ut, iaculis quis, sem. Suspendisse sagittis ultrices augue. Donec ipsum massa, ullamcorper in, auctor et, scelerisque sed, est. In sem justo, commodo ut, suscipit at, pharetra vitae, orci. Pellentesque pretium lectus id turpis.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Curabitur sagittis hendrerit ante. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Cras pede libero, dapibus nec, pretium sit amet, tempor quis. Sed vel lectus. Donec odio tempus molestie, porttitor ut, iaculis quis, sem. Suspendisse sagittis ultrices augue. Donec ipsum massa, ullamcorper in, auctor et, scelerisque sed, est. In sem justo, commodo ut, suscipit at, pharetra vitae, orci. Pellentesque pretium lectus id turpis.

In Prague on December 29, 2025

Abstract

R is a widely-used dynamic language for statistical computing, but its dynamic nature prevents efficient ahead-of-time compilation. We present the design and implementation of a compiler for a statically-typed subset of R that targets WebAssembly. The defined language retains R's core characteristics, including first-class vector operations and lexical scoping, while introducing static typing to enable compilation. The compiler leverages the WebAssembly Garbage Collection proposal, which simplifies memory management and enables efficient representation of high-level language constructs. Source programs are transformed through parsing, type checking, and intermediate representation lowering to generate WebAssembly bytecode, supported by a runtime system providing vector operations and host environment interoperability. Evaluation on representative statistical programs demonstrates the viability of compiling R-like languages to WebAssembly using modern proposals.

Keywords WebAssembly, WASM GC, typed R, WASM bytecode, pass-based Compiler

Abstrakt

Fill in the abstract of this thesis in Czech. Lorem ipsum dolor sit amet. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Cras pede libero, dapibus nec, pretium sit amet, tempor quis. Sed vel lectus. Donec odio tempus molestie, porttitor ut, iaculis quis, sem. Suspendisse sagittis ultrices augue.

Klíčová slova enter, comma, separated, list, of, keywords, in, CZECH

Contents

List of Figures

List of Tables

List of Code listings

List of abbreviations

WASM	Web Assembly
WAT	Web Assembly Text
GC	Garbage Collector
IR	Intermediate Representation

Chapter 1
Introduction

The R programming language is ubiquitous for statistical computing and data analysis, offering powerful abstraction for data manipulation and visualization. Inspired by S-Language, the R language has been important part for industries working with statistics and been pioneer for introducing widespread tools currently used by the community such as DataFrame.

Traditionally, R is an interpretered language that runs on most major operating systems. By interpretered language, we mean that the language source code is not translated to certain binary to be executed by processor like compiled languages. Instead interpreter is a program, usually written in low-level languages such as C, C++ or Rust, which takes the source code, does lexing and parsing to get AST and directly executes the program from its AST representation by evaluating from top node[1]. Figure 1.1 shows overview of R code being interpretted.

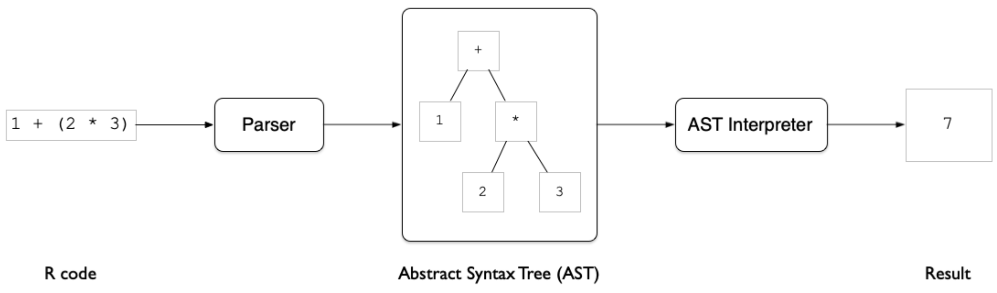

Figure 1.1 Overview. Taken from R Bytecode Book website

Nowadays, the industry has been expanding the environments where we can run our programs. The interpreter program we discussed about, is compiled to bytecode, and then binary that the processor can execute. However, with invention of WebAssembly, we can take this interpreter and compile it to so-called WASM bytecode, and run it on web. This is exactly what webR[2] did. This way, R code we write, can seamlessly run on the web environment on most browsers.

Compiling the interpreter to WASM introduces certain challenges. Since the interpreter consists of a substantial amount of code, every R program executed in the browser must include the entire interpreter compiled to WASM. This increases the overall memory footprint and can negatively impact performance, particularly in resource-constrained environments.

Thus, this thesis explores direct compilation of R-like language to WebAssembly. I create subset of R with type annotations, in order to make compilation possible to WASM, as it needs static typing. Then explore the challenges of mapping a dynamic high-level language to WASM, and finally evaluate the performance and correctness of my compiler.

Chapter 2

Background

2.1 The R Programming Language

R is a domain-specific language designed for statistical computing and graphics, originally developed by Ross Ihaka and Robert Gentleman in the early 1990s as an open-source implementation of the **S** language. With its CRAN package manager providing more than 20000 packages, it's a widely programming language used by industries where experimenting with data is involved, such as data analytics, data mining and bio-informatics.

2.1.1 Key Characteristics

- **Dynamic typing:** Variables have no declared types; types are determined at runtime. This enables rapid prototyping but prevents static verification and certain compiler optimizations.
- **Vectorization:** Operations apply element-wise to vectors, matrices, and arrays. For example,

  `c(1,2,3) + c(4,5,6)` $\Rightarrow$ `c(5,7,9)`

  without the need for explicit loops.
- **Lexical scoping:** R uses lexical (static) scoping with closures. Functions capture their defining environment, enabling functional programming patterns and higher-order abstractions.
- **Lazy evaluation:** Function arguments are evaluated lazily (call-by-need), allowing non-standard evaluation mechanisms that support domain-specific sublanguages.
- **Copy-on-modify semantics:** R employs implicit copying to preserve referential transparency. While this simplifies reasoning about programs, it may introduce performance overhead in memory-intensive workloads.

2.1.2 Assignment Operators

R provides multiple assignment operators with distinct scoping behavior:

- `<-`: Regular assignment in the current environment.
- `<<-`: Superassignment, which modifies the nearest existing binding in an enclosing environment.

2.1.3 Type System

R employs a dynamic type system with several foundational types, all of which are first-class objects. Unlike statically-typed languages, type information is associated with values at runtime rather than with variable declarations. Atomic Types R provides six atomic vector types:

- **Logical:** Boolean values `TRUE`, `FALSE`, and `NA` (missing).

  ```
  x <- TRUE
  typeof(x)      # returns "logical"
  ```

- **Integer:** Whole numbers, denoted with an `L` suffix.

  ```
  x <- 42L
  typeof(x)      # returns "integer"
  ```

- **Double:** Floating-point numbers (default numeric type).

  ```
  x <- 3.14
  typeof(x)      # returns "double"
  ```

- **Character:** Strings of text.

  ```
  x <- "hello"
  typeof(x)      # returns "character"
  ```

- **Complex:** Complex numbers with real and imaginary parts.

  ```
  x <- 2 + 3i
  typeof(x)      # returns "complex"
  ```

- **Raw:** Raw bytes (rarely used).

  ```
  x <- charToRaw("A")
  typeof(x)      # returns "raw"
  ```

Composite Types Beyond atomic vectors, R supports several composite data structures:

- **List:** Heterogeneous collections that can hold elements of different types.

  ```
  x <- list(42, "text", TRUE)
  typeof(x)      # returns "list"
  ```

- **Function:** Functions are first-class objects.

  ```
  f <- function(x) x + 1
  typeof(f)      # returns "closure"
  ```

  This means functions are treated as normal variables. It can be nested definition, passed as parameter and returned from a function.

- **Environment:** Hash-like structures for variable scoping.

  ```
  e <- new.env()
  typeof(e)      # returns "environment"
  ```

Type Coercion R performs implicit type coercion following a hierarchy: logical → integer → double → character. When combining types, R automatically converts to the most general type:

```
c(TRUE, 1L, 2.5, "text")  # returns character vector:
# "TRUE" "1" "2.5" "text"
```

This automatic coercion simplifies interactive use but can lead to unexpected behavior if types are not carefully managed.

2.1.4 Peculiarities

As every programming language comes with their nuances and uniqueness, R is no short of those. Below, the ones mention worthy examples

- Vectors and lists are indexed by 1. For example,

  ```
          v <- c(10,20,30) // initializes a vector of 10,20,30
          v[1]             // returns 10
          v[2]             // returns 20
  ```

- Even a scalar is represented as a vector of 1 element.

  ```
          x <- 5           // assign 5
          is.vector(x)     // return TRUE
  ```

The reason for it is to recycle the vectors easily. The language is designed for vector operations

```
x <- 5           // assign 5
v <- c(10,20,30) // initializes a vector of 10,20,30
x + v            // returns a vector of 15,25,35
```

2.2 WebAssembly

WebAssembly is a low-level assembly-like language that is made to run in modern browser[3]. It's designed to work together with JavaScript, the language of the web environment, offering flexibility and performance. Currently most modern languages support WASM as compilation target; for example, Rust through rustc, C/C++ through Emscripten.

Practically, WebAssembly offers myriad of possibilities to developers. From Image-processing libraries in hidden behind C to desktop apps written in C#, with WASM , they can run on the web with near native-like performance. Notable examples of adoption of WASM are Figma[4], Autodesk AutoCAD Web App[5], and Google Earth[6].

Moreover, for readability, WebAssembly also has so-called Web Assembly Text Format, in short WAT. It's a form where bytecode itself is more readable, with more syntactic sugars. Let's iterate over properties of WASM and how it's structured and written in WAT format.

Module system WebAssembly code is organized into modules. Modules declare imports (functions, memory, tables, globals from the host environment) and exports (making internal definitions available externally)[7]:

```
(module
  ;; Import a logging function from the host
  (import "env" "log" (func $log (param i32)))

  ;; Import memory from the host
  (import "env" "memory" (memory 1))

  ;; Internal function (not exported)
  (func $internal_add (param $a i32) (param $b i32) (
     result i32)
    local.get $a
    local.get $b
    i32.add
  )

  ;; Can write memory
  ;; Can create data segment

  ;; Public function that uses imports
  (func $add_and_log (param $a i32) (param $b i32) (result i32)
    local.get $a
    local.get $b
    call $internal_add     ;; call internal function
    local.get 0            ;; duplicate result for logging
    call $log              ;; log the result
  )

  ;; Export the public function
  (export "addAndLog" (func $add_and_log))
)
```

Code listing 2.1 Module demonstrating import and export mechanisms

Stack-based machine With goals of being fast, efficient, and portable, WebAssembly, at its heart, is a stack-based virtual machine running on the web. Stack-based machine in this context is a process virtual machine that works as virtualization of computer system on top of a computer. Its workings is primarily based on interacting with the stack. For example:

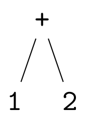

Figure 2.1 Tree representation

```
(module
  (func $compute (result i32)
    i32.const 1        ;; push constant 1 onto stack
    i32.const 2        ;; push constant 2 onto stack
    i32.add            ;; pop two values, push sum
  )
  (export "compute" (func $compute))
)
```

Code listing 2.2 A simple WebAssembly function showing stack operations

Structured control flow Unlike raw bytecode with goto-style jumps, WebAssembly uses structured control flow constructs: block, loop, if, and br (branch). Each construct creates a label that branches can target.[7]

```
(module
  (func $max (param $a i32) (param $b i32) (result i32)
    local.get $a
    local.get $b
    i32.gt_s                  ;; signed greater-than
       comparison
    (if (result i32)
      (then
        local.get $a
      )
      (else
        local.get $b
      )
    )
  )
  (export "max" (func $max))
)
```

Code listing 2.3 Conditional execution using structured control flow

Linear memory model WebAssembly provides a contiguous, resizable array of bytes called linear memory. Memory is accessed via load/store instructions with explicit alignment and offset.[7]

```
(module
  (memory 1)                  ;; declare 1 page (64KB) of
     memory
  (func $increment_at_zero
    i32.const 0               ;; memory address
    i32.const 0               ;; memory address (for load)
    i32.load                  ;; load 32-bit value from
       address 0
    i32.const 1               ;; constant 1
    i32.add                   ;; increment
    i32.store                 ;; store result back to address
        0
  )
  (export "memory" (memory 0))
  (export "increment" (func $increment_at_zero))
)
```

Code listing 2.4 Accessing linear memory with load and store instructions

Type system and Determinism A decoded module is type-safe and checked in module instantiation. Moreover, WebAssembly specification fully defines valid programs and their behaviour.[7]

Chapter 3

Language Design and Mappings

In this chapter, we'll design a subset of R with types and see how types and their operations could map to WASM generation.

3.1 Design Goals

To design typed R, I tried to be as close as possible to R semantics to easily compile available R codes with wasmR. Although there are obviously inherent limitations compiling dynamic language to statically typed bytecode. Issues include mutability, typing, reflections. For example, how do we deal with

```
var <- 21  // assign integer
print(var)
var <- c(1,2,3) // assign vector
```

This is a perfectly fine code for R but compiling it to WASM brings difficulties. Therefore, like other static languages, we will add checking against mutability of different types.

- R compatibility: Preserve R's syntax and core semantics where possible
- Type safety: Static type checking with sound type system
- Ahead-of-time compilation
- A single executable WASM file, no linking required
- First-class functions: Support functional programming with closures
- Practicality: Provide essential functions for data processing (builtins)

3.2 Typed R-like Language

This section presents the design of a statically-typed programming language inspired by R's syntax, which we refer to as the *Typed R-like Language*. The language maintains R's characteristic features such as the left-assignment operator (`<-`), first-class functions, and vector-oriented programming, while introducing a static type system to enable ahead-of-time compilation to WebAssembly.

The language supports:

- **Static typing** with type inference and explicit type annotations
- **First-class functions** with closures and lexical scoping
- **Vector operations** as a fundamental data structure
- **Control flow** constructs including conditionals and loops
- **Higher-order functions** enabling functional programming patterns
- **Lexical scoping** with support for nested functions and variable capture
- **Named arguments** enabling option to have positional and optional arguments

The design philosophy emphasizes a familiar R-like syntax while ensuring type safety and efficient compilation to WebAssembly. Unlike dynamically-typed R, all type information is resolved at compile time, enabling optimized code generation and early error detection.

3.3 Syntax

The syntax of the Typed R-like Language closely follows R conventions with extensions for explicit type annotations. This section describes the core syntactic constructs.

$$\begin{array}{rcl}
\langle \textit{program} \rangle & ::= & \langle \textit{statement} \rangle^{*} \\
\langle \textit{statement} \rangle & ::= & \langle \textit{assignment} \rangle \mid \langle \textit{control flow} \rangle \mid \langle \textit{expression} \rangle \mid \langle \textit{index assign} \rangle \mid \langle \textit{return} \rangle \\
\langle \textit{control flow} \rangle & ::= & \langle \textit{if} \rangle \mid \langle \textit{for} \rangle \mid \langle \textit{while} \rangle \\
\langle \textit{if} \rangle & ::= & \texttt{if (}\langle \textit{expression} \rangle\texttt{) }\{\langle \textit{statements} \rangle\} \\
 & & \mid \texttt{if (}\langle \textit{expression} \rangle\texttt{) }\{\langle \textit{statements} \rangle\}\texttt{ else }\{\langle \textit{statements} \rangle\} \\
\langle \textit{for} \rangle & ::= & \texttt{for (}\langle \textit{identifier} \rangle\texttt{ in }\langle \textit{expression} \rangle\texttt{) }\{\langle \textit{statements} \rangle\} \\
\langle \textit{assignment} \rangle & ::= & \langle \textit{identifier} \rangle\ [\texttt{:}\langle \textit{type} \rangle]\ \texttt{<-}\ \langle \textit{expression} \rangle \\
\langle \textit{expression} \rangle & ::= & \langle \textit{literal} \rangle \mid \langle \textit{identifier} \rangle \mid \langle \textit{function\_def} \rangle \mid \langle \textit{function\_call} \rangle \mid \ldots \\
\langle \textit{function\_def} \rangle & ::= & \texttt{function (}\langle \textit{param\_list} \rangle\texttt{) :}\langle \textit{type} \rangle\ \{\langle \textit{statement} \rangle^{*}\} \\
\langle \textit{param\_list} \rangle & ::= & \langle \textit{param} \rangle\ \texttt{(,}\langle \textit{param} \rangle\texttt{)}^{*} \\
\langle \textit{param} \rangle & ::= & \langle \textit{identifier} \rangle\texttt{:}\langle \textit{type} \rangle \\
\langle \textit{function\_call} \rangle & ::= & \langle \textit{identifier} \rangle\texttt{(}\langle \textit{arg\_list} \rangle\texttt{)} \\
\langle \textit{function\_call} \rangle & ::= & \langle \textit{identifier} \rangle\texttt{(}\langle \textit{arg\_list} \rangle\texttt{)} \\
\langle \textit{arg\_list} \rangle & ::= & \langle \textit{expression} \rangle\ \texttt{(,}\langle \textit{expression} \rangle\texttt{)}^{*} \\
\langle \textit{type} \rangle & ::= & \texttt{integer} \mid \texttt{logical} \mid \texttt{double} \mid \texttt{(}\langle \textit{type} \rangle\texttt{ -> }\langle \textit{type} \rangle\texttt{)} \\
 & & \mid \texttt{any} \mid \texttt{function} \mid \langle \textit{composite type} \rangle \\
\langle \textit{composite type} \rangle & ::= & \texttt{vector}\ \langle \textit{type} \rangle
\end{array}$$

3.3.1 Lexical Elements

The language uses the following lexical tokens:

- **Keywords:** `function`, `if`, `else`, `for`, `in`, `while`, `return`
- **Type keywords:** `int`, `double`, `void`, `logical`, `any`, `vector`, `list`
- **Operators:**
  - Assignment: `<-` (assignment), `<<-` (super-assignment)
  - Arithmetic: `+`, `-`, `*`, `/`, `%%` (modulo)
  - Comparison: `==`, `!=`, `<`, `<=`, `>`, `>=`
  - Logical: `&` (and), `|` (or), `!` (not)
  - Range: `:` (sequence generation)

  - Type annotation: `:` (in declaration context)
  - Function arrow: `->` (in type signatures)
- **Delimiters:** `(`, `)`, `{`, `}`, `[`, `]`, `,`
- **Literals:** Numeric literals, string literals (double-quoted), logical literals (`TRUE`, `FALSE`)
- **Identifiers:** Alphanumeric sequences starting with a letter or underscore
- **Special:** `...` (varargs placeholder)

3.3.2 Variable Declaration and Assignment

Variables are declared using the left-assignment operator with optional type annotations:

```
# Simple assignment with type inference
x <- 10

# Assignment with explicit type annotation
y: int <- 20

# Vector assignment
vec <- c(1, 2, 3)
```

Code listing 3.1 Variable assignment examples

```
(module
  ;; Define array type for vectors
  (type $vec_i32 (array (mut i32)))

  (func $main
    (local $x i32)
    (local $y i32)
    (local $vec (ref null $vec_i32))

    ;; x <- 10
    i32.const 10
    local.set $x

    ;; y: int <- 20
    i32.const 20
    local.set $y

    ;; vec <- c(1, 2, 3)
    i32.const 1
    i32.const 2
    i32.const 3
    i32.const 3
    array.new_fixed $vec_i32 3
    local.set $vec
  )

  (start $main)
)
```

■ **Code listing 3.2** Variable assignment in WAT

The super-assignment operator `<<-` modifies variables in enclosing function scopes:

```
outer <- function() {
    x <- 0
    inner <- function() {
        x <<- 10  # Modifies x in outer scope
    }
    inner()
    return(x)  # Returns 10
}
```

■ **Code listing 3.3** Super-assignment example

This takes certain workaround to compile it in WASM. We'll talk about its compilation in further chapters.

3.3.3 Function Definitions

Functions are first-class values defined using the `function` keyword:

```
# Simple function with type annotations
add <- function(a: int, b: int): int {
    return(a + b)
}

# Function returning a vector
create_vector <- function(): vector<double> {
    return(c(1.0, 2.0, 3.0))
}

# Higher-order function
apply_twice <- function(f: int -> int, x: int): int {
    return(f(f(x)))
}
```

■ **Code listing 3.4** Function definition examples in typed R

```
(module
  ;; Define array type for double vectors
  (type $vec_f64 (array (mut f64)))

  ;; add <- function(a: int, b: int): int
  (func $add (param $a i32) (param $b i32) (result i32)
    local.get $a
    local.get $b
    i32.add
  )

  ;; create_vector <- function(): vector<double>
  (func $create_vector (result (ref $vec_f64))
    f64.const 1.0
    f64.const 2.0
    f64.const 3.0
    i32.const 3
    array.new_fixed $vec_f64 3
  )
  ;; apply_twice is not shown here.

)
```

Code listing 3.5 Function definition examples in WAT

Function types use arrow notation: `param_types -> return_type`. Multiple parameters are comma-separated, and parentheses group complex function types when used as parameters.

3.3.4 Control Flow

3.3.4.1 Conditional Statements

The language supports `if-else` statements and expressions:

```
# If statement
if (x > 0) {
    print(x)
}

# If-else statement
if (x > 0) {
    y <- 1
} else {
    y <- -1
}

# If expression (returns value)
result <- if (x > 0) { 1 } else { -1 }
```

Code listing 3.6 Conditional examples in typed R

```
(module
  ;; Import print function from host environment
  (import "env" "print" (func $print (param i32)))

  (func $main
    (local $x i32)
    (local $y i32)

    ;; Set x to some value for demonstration
    i32.const 5
    local.set $x

    ;; if (x > 0) { print(x) }
    local.get $x
    i32.const 0
    i32.gt_s                ;; x > 0 (signed comparison)
    if
      local.get $x
      call $print
    end

    ;; if (x > 0) { y <- 1 } else { y <- -1 }
    local.get $x
    i32.const 0
    i32.gt_s                ;; x > 0
    if
      i32.const 1
      local.set $y
    else
      i32.const -1
      local.set $y
    end
  )

  (start $main)
)
```

Code listing 3.7 Conditional examples in WAT

3.3.4.2 Loops

Two loop constructs are provided: `for` and `while`.

```
# For loop iterating over range
for (i in 1:10) {
    print(i)
}

# For loop iterating over vector
vec <- c(1, 2, 3, 4, 5)
for (x in vec) {
    print(x)
}

# While loop
i <- 1
sum <- 0
while (i <= 5) {
    sum <- sum + i
    i <- i + 1
}
```

Code listing 3.8 Loop examples

```
(module
  (func $main
    (local $i i32)
    (local $sum i32)

    ;; i <- 1
    i32.const 1
    local.set $i

    ;; sum <- 0
    i32.const 0
    local.set $sum

    ;; while (i <= 5) { ... }
    (block $break
      (loop $continue
        ;; Check condition: i <= 5
        local.get $i
        i32.const 5
        i32.le_s                ;; i <= 5 (signed comparison)
        i32.eqz                 ;; negate: if NOT (i <= 5)
        br_if $break            ;; break if condition is
            false

        ;; sum <- sum + i
        local.get $sum
        local.get $i
        i32.add
        local.set $sum

        ;; i <- i + 1
        local.get $i
        i32.const 1
        i32.add
        local.set $i

        ;; Continue loop
        br $continue
      )
    )
  )

  (start $main)
)
```

Code listing 3.9 While loop examples in WAT

3.3.5 Expressions

Abstract syntax tree of expressions

$$
\begin{array}{lll}
e ::= x & & \text{(variable)} \\
\quad \mid n & & \text{(literal)} \\
\quad \mid e_1 \; op \; e_2 & & \text{(binary operation)} \\
\quad \mid op \; e_1 & & \text{(unary operation)} \\
\quad \mid \texttt{if} \; e_1 \; \texttt{then} \; e_2 \; \texttt{else} \; e_3 & & \text{(conditional)} \\
\quad \mid \texttt{fun}(x : \tau) \to e & & \text{(function)} \\
\quad \mid e_1(e_2) & & \text{(application)}
\end{array}
$$

The language supports various expression forms:

- **Literals:** `42L`, `3 3.14`, `TRUE`, `FALSE`
- **Identifiers:** Variable references
- **Binary operations:** Arithmetic, comparison, logical, range (`1:10`)
- **Unary operations:** Negation (`-x`), logical not (`!x`)
- **Function calls:** `f(arg1, arg2)` with positional or named arguments
- **Vector indexing:** `vec[i]`
- **Vector construction:** `c(1, 2, 3)`
- **Function definitions:** Anonymous functions as expressions
- **If expressions:** Conditional expressions returning values

3.3.6 Blocks and Tail Expressions

Blocks consist of zero or more statements followed by an optional tail expression. The tail expression (final expression without semicolon) determines the block's value:

```
f <- function(x: int): int {
    y <- x * 2
    z <- y + 1
    z  # Tail expression - returned automatically
}
```

Code listing 3.10 Block with tail expression

3.4 Type System

The type system ensures type safety through static analysis while supporting type inference to maintain concise syntax. Though it's important to note that there are many programs where type inference will fail to identify the type and our compiler won't work due to lack of type information. (reference needed as to why type inference is hard)

$$
\begin{aligned}
\tau &::= \mathit{num} \mid \texttt{void} && \text{(base type)}\\
&\mid \tau\,\texttt{[]} && \text{(vector type)}\\
&\mid (\tau_1, \ldots, \tau_n) \to \tau && \text{(function type)}\\
\mathit{num} &::= \texttt{int} \mid \texttt{double} \mid \texttt{bool} && \text{(numeric types)}
\end{aligned}
$$

3.4.1 Primitive Types

The language provides the following primitive types:

- `int`: 32-bit signed integers (maps to WebAssembly `i32`)
- `double`: 64-bit floating-point numbers (maps to WebAssembly `f64`)
- `logical`: Boolean values (`TRUE` or `FALSE`)
- `void`: Absence of value (used for functions with no return)

Logical Logicals will be represented as `int32` in WebAssembly, as it doesn't provide explicit type for logicals there.

3.4.2 Composite Types

3.4.2.1 Vector Types

Vectors are homogeneous arrays parameterized by element type:

```
# Vector of integers
v1: vector<int> <- c(1, 2, 3)

# Vector of doubles
v2: vector<double> <- c(1.5, 2.5, 3.5)

# Vector operations (component-wise)
v3: vector<double> <- v2 + c(1.0, 2.0, 3.0)
```

Code listing 3.11 Vector type examples

Vectors support component-wise arithmetic operations when both operands have compatible vector types.

3.4.2.2 Function Types

Function types represent callable values with parameter and return types:

```
<type> ::= <function_type>

<function_type> ::= <param_list> "->" <type>
                  | <primary_type>

<param_list> ::= <primary_type>
               | <primary_type> "," <param_list>

<primary_type> ::= <builtin_type>
                 | <generic_type>
                 | "(" <function_type> ")"
                 | "function"

<builtin_type> ::= "int" | "double" | "string" | "char"
                 | "void" | "logical" | "any"

<generic_type> ::= "vector" "<" <type> ">"
                 | "list" "<" <type> ">"
```

Examples of function types:

- `int -> int`: Function taking an integer, returning an integer
- `int, int -> double`: Function taking two integers, returning a double
- `(int -> int) -> int`: Higher-order function taking a function as parameter
- `int, (int, int -> int) -> int`: Function taking an integer and a function

3.4.3 Type Inference

The type checker performs bidirectional type inference:

- **Bottom-up inference:** Expression types are inferred from literal values and operator signatures
- **Top-down checking:** Function return types and variable annotations provide expected types
- **Unification:** Type constraints are solved to determine concrete types

Type annotations are required in the following contexts:

- Function parameters
- Function return types (when not inferrable from return statements)
- Ambiguous variable declarations

3.4.4 Typing Rules

Selected typing rules are presented below using inference rule notation.

3.4.4.1 Variable Reference

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T}$$

A variable reference has the type bound in the environment Γ.

3.4.4.2 Function Application

$$\frac{\Gamma \vdash e_1 : T_1, \ldots, T_n \to T \quad \Gamma \vdash e_2 : T_1 \quad \cdots \quad \Gamma \vdash e_{n+1} : T_n}{\Gamma \vdash e_1(e_2, \ldots, e_{n+1}) : T}$$

Function application checks that argument types match parameter types and produces the return type.

3.4.4.3 Function Definition

$$\frac{\Gamma, x_1 : T_1, \ldots, x_n : T_n \vdash e : T_{ret}}{\Gamma \vdash \texttt{function}(x_1 : T_1, \ldots, x_n : T_n) : T_{ret}\{e\} : (T_1, \ldots, T_n \to T_{ret})}$$

A function definition has a function type where parameters and return type match the declared signature.

3.4.4.4 Binary Operations

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 + e_2 : \texttt{int}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{double} \quad \Gamma \vdash e_2 : \texttt{double}}{\Gamma \vdash e_1 + e_2 : \texttt{double}}$$

Arithmetic operators are overloaded for numeric types. Comparison operators produce `logical` values.

3.4.4.5 Vector Operations

$$\frac{\Gamma \vdash e_1 : \texttt{vector} < T > \quad \Gamma \vdash e_2 : \texttt{vector} < T >}{\Gamma \vdash e_1 + e_2 : \texttt{vector} < T >}$$

Component-wise vector operations require matching element types.

3.4.4.6 Conditionals

$$\frac{\Gamma \vdash e_{cond} : \texttt{logical} \quad \Gamma \vdash e_{then} : T \quad \Gamma \vdash e_{else} : T}{\Gamma \vdash \texttt{if } (e_{cond})\ \{e_{then}\}\ \texttt{else}\ \{e_{else}\} : T}$$

If-expressions require a logical condition and both branches must have the same type.

3.4.5 Scoping and Closures

The language uses lexical scoping where:

- Only functions create new scopes (blocks, if-statements, and loops do not)
- Functions can capture variables from enclosing function scopes
- Captured variables are implemented using closure environments in WebAssembly
- Super-assignment (`<<-`) searches enclosing function scopes to modify variables

Example demonstrating closure:

```
make_counter <- function(start: int): () -> int {
    count <- start
    function(): int {
        count <<- count + 1
        return(count)
    }
}

counter <- make_counter(0)
```

```
print(counter())  # Prints 1
print(counter())  # Prints 2
```

Code listing 3.12 Closure example

3.5 Semantics

This section describes the operational semantics of key language constructs.

3.5.1 Expression Evaluation

Expression evaluation follows a small-step operational semantics with evaluation contexts.

3.5.1.1 Literal Evaluation

Literals evaluate to themselves:

$$\begin{aligned} n \Downarrow n &\quad \text{(numeric literal)} \\ \texttt{"str"} \Downarrow \texttt{"str"} &\quad \text{(string literal)} \\ \texttt{TRUE} \Downarrow \texttt{TRUE} &\quad \text{(logical literal)} \end{aligned}$$

3.5.1.2 Variable Lookup

Variable references evaluate by environment lookup:

$$\frac{\rho(x) = v}{\langle x, \rho \rangle \Downarrow v}$$

where ρ is the runtime environment mapping identifiers to values.

3.5.1.3 Binary Operations

Binary operations evaluate operands left-to-right, then apply the operation:

$$\frac{\langle e_1, \rho \rangle \Downarrow v_1 \quad \langle e_2, \rho \rangle \Downarrow v_2 \quad v_1 \oplus v_2 = v}{\langle e_1 \oplus e_2, \rho \rangle \Downarrow v}$$

where $\oplus$ represents any binary operator.

Arithmetic Operators (`+`, `-`, `*`, `/`, `%%`) on same types:

$$\begin{aligned}
n_1 + n_2 &= n_1 + n_2 \quad (\text{int} + \text{int} \to \text{int}) \\
d_1 + d_2 &= d_1 + d_2 \quad (\text{double} + \text{double} \to \text{double}) \\
n_1 - n_2 &= n_1 - n_2 \\
n_1 * n_2 &= n_1 \times n_2 \\
n_1 / n_2 &= \lfloor n_1 \div n_2 \rfloor \quad (\text{integer division}) \\
d_1 / d_2 &= d_1 \div d_2 \quad (\text{floating-point division}) \\
n_1 \bmod n_2 &= n_1 \bmod n_2
\end{aligned}$$

Type Promotion: When operands have different numeric types, implicit casting promotes to the wider type:

$$\begin{aligned}
\text{int} \oplus \text{double} &\to \text{double} \\
\text{logical} \oplus \text{int} &\to \text{int} \\
\text{logical} \oplus \text{double} &\to \text{double}
\end{aligned}$$

Example: `3 + 2.5` evaluates as `3.0 + 2.5 = 5.5` after promoting `3` to `double`.

Comparison Operators (`==`, `!=`, `<`, `<=`, `>`, `>=`) produce logical values:

$$\begin{aligned}
n_1 < n_2 &= \texttt{TRUE} \text{ if } n_1 < n_2, \text{ else } \texttt{FALSE} \\
n_1 == n_2 &= \texttt{TRUE} \text{ if } n_1 = n_2, \text{ else } \texttt{FALSE}
\end{aligned}$$

Type promotion applies: `3 < 2.5` compares as `3.0 < 2.5`.

Logical Operators (`&`, `|`) on boolean values:

$$\begin{aligned}
b_1 \;\texttt{\&}\; b_2 &= b_1 \wedge b_2 \\
b_1 \;\texttt{|}\; b_2 &= b_1 \vee b_2
\end{aligned}$$

Vector Operations are applied component-wise after ensuring compatible types:

$$\texttt{vector}[v_1, \ldots, v_n] \oplus \texttt{vector}[w_1, \ldots, w_n] = \texttt{vector}[v_1 \oplus w_1, \ldots, v_n \oplus w_n]$$

For mixed vector types, element types are promoted (e.g., `vector<int>` + `vector<double>` produces `vector<double>`).

Vector-Scalar Operations: Scalars are broadcast to match vector length:

$$\texttt{vector}[v_1, \ldots, v_n] \oplus s = \texttt{vector}[v_1 \oplus s, \ldots, v_n \oplus s]$$

3.5.1.4 Function Application

Function application evaluates the callee to obtain a closure, evaluates arguments, extends the closure environment, and evaluates the body:

$$\frac{\begin{array}{l}\langle e_0, \rho\rangle \Downarrow \langle \lambda(x_1, \ldots, x_n).e, \rho'\rangle \\ \langle e_1, \rho\rangle \Downarrow v_1 \quad \cdots \quad \langle e_n, \rho\rangle \Downarrow v_n \\ \langle e, \rho'[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]\rangle \Downarrow v\end{array}}{\langle e_0(e_1, \ldots, e_n), \rho\rangle \Downarrow v}$$

3.5.2 Statement Execution

Statements modify the environment and may alter control flow.

3.5.2.1 Assignment

Assignment evaluates the right-hand side and binds the result to the identifier:

$$\frac{\langle e, \rho\rangle \Downarrow v}{\langle x \ \texttt{<-}\ e, \rho\rangle \to \rho[x \mapsto v]}$$

Super-assignment searches parent scopes:

$$\frac{\langle e, \rho\rangle \Downarrow v \quad x \in \mathrm{dom}(\rho_{parent})}{\langle x \ \texttt{<<-}\ e, \rho\rangle \to \rho[\rho_{parent}[x \mapsto v]]}$$

3.5.2.2 Conditional Execution

If-statements evaluate the condition and execute the appropriate branch:

$$\frac{\langle e_{cond}, \rho\rangle \Downarrow \texttt{TRUE} \quad \langle s_{then}, \rho\rangle \to \rho'}{\langle \texttt{if}\ (e_{cond})\ \{s_{then}\}, \rho\rangle \to \rho'}$$

$$\frac{\langle e_{cond}, \rho\rangle \Downarrow \texttt{FALSE} \quad \langle s_{else}, \rho\rangle \to \rho'}{\langle \texttt{if}\ (e_{cond})\ \{s_{then}\}\ \texttt{else}\ \{s_{else}\}, \rho\rangle \to \rho'}$$

3.5.2.3 While Loops

While loops repeatedly execute the body while the condition is true:

$$\frac{\langle e_{cond}, \rho\rangle \Downarrow \texttt{FALSE}}{\langle \texttt{while}\ (e_{cond})\ \{s\}, \rho\rangle \to \rho}$$

$$\frac{\begin{array}{l}\langle e_{cond}, \rho\rangle \Downarrow \texttt{TRUE} \\ \langle s, \rho\rangle \to \rho' \\ \langle \texttt{while}\ (e_{cond})\ \{s\}, \rho'\rangle \to \rho''\end{array}}{\langle \texttt{while}\ (e_{cond})\ \{s\}, \rho\rangle \to \rho''}$$

3.5.2.4 For Loops

For loops iterate over vectors or ranges:

$$\frac{\begin{array}{l}\langle e_{vec}, \rho\rangle \Downarrow \texttt{vector}[v_1, \dots, v_n] \\ \langle s, \rho[x \mapsto v_1]\rangle \to \rho_1 \\ \dots \\ \langle s, \rho_{n-1}[x \mapsto v_n]\rangle \to \rho_n\end{array}}{\langle \texttt{for}\ (x\ \texttt{in}\ e_{vec})\ \{s\}, \rho\rangle \to \rho_n}$$

3.5.3 Function Calls and Return

Function calls push a new activation frame onto the call stack. The `return` statement terminates function execution and yields a value:

$$\frac{\langle e, \rho\rangle \Downarrow v}{\langle \texttt{return}(e), \rho\rangle \Rightarrow v}$$

The special $\Rightarrow$ arrow indicates early exit from the function body.

3.5.4 Built-in Functions

The language provides built-in functions with special semantics:

- `c(e1, ..., en)`: Creates a vector from arguments. All arguments must have the same type.
- `print(e)`: Outputs the value of expression `e` to standard output.
- `length(v)`: Returns the number of elements in vector `v`.
- `sum(v)`: Returns the sum of all elements in numeric vector `v`.
- `gen_seq(start, end)`: Generates a sequence from `start` to `end` (equivalent to `start:end`).

Chapter 4

Compiler and Runtime

4.1 Compiler Architecture

Frontend, type checking, intermediate representation, and backend.

4.2 WebAssembly Code Generation

Translation of language constructs and types to WebAssembly.

4.3 Runtime System

Vector representation, memory management, and built-in operations.

4.4 Implementation Details

Key implementation choices and limitations.

Chapter 5
Evaluation

5.1 Correctness

Testing methodology and comparison with R behavior.

5.2 Performance

Execution time, code size, and runtime overhead.

5.3 Discussion

Analysis of results and trade-offs.

Chapter 6

Conclusion

Summary of contributions and future work.

Appendix A
Some content

An example below to show how the bytecode for generic functions would look like in arbitrary typed bytecode. How would we compile untyped code for?

1. The high level code

   ```
   c = a + b
   ```

2. Every value is a boxed value.

   ```
   Value {
     tag : TypeTag
     payload : union {
       int64
       float64
       pointer
       object_ref
     }
   }

   TypeTags :== INT | FLOAT | STRING | OBJECT | ...
   ```

3. Load variable and call the generic function (simplified instruction)

   ```
   LOAD_LOCAL   a
   LOAD_LOCAL   b
   ADD_GENERIC
   STORE_LOCAL  c
   ```

4. What ADD_GENERIC have to do?

   ```
   b = pop()
   a = pop()

   if a.tag == INT and b.tag == INT:
   ```

```
        push(int_add(a, b))
    elif a.tag == FLOAT and b.tag == FLOAT:
        push(float_add(a, b))
    elif a.tag == STRING and b.tag == STRING:
        push(string_concat(a, b))
    elif a.tag == OBJECT:
        call a.__add__(b)
    else:
        runtime_type_error()
```

Remember, this is optimistic scenario. What happens when we have on LHS(left-hand side) an INT and on RHS(right-hand side) FLOAT? Moreover what if one of them is composite types? As one can see this dispatcher function for every operation combinated with every type will be a huge overhead.

Bibliography

1. NYSTROM, Robert. *Crafting Interpreters*. Genever Benning, 2021. ISBN 978-0990582939. Available also from: https://craftinginterpreters.com/.
2. RSTUDIO, PBC. *webR*. 2023. Version 1.0. Available also from: https://webr.rstudio.com. Accessed: 2025-12-28.
3. MDN WEB DOCS. *WebAssembly*. 2025. Available also from: https://developer.mozilla.org/en-US/docs/WebAssembly. Accessed: 2025-12-28.
4. EVAN WALLACE. *WebAssembly cut Figma's load time by 3x*. 2017. Available also from: https://www.figma.com/blog/webassembly-cut-figmas-load-time-by-3x/. Accessed: 2025-12-28.
5. AARON TURNER. *AutoCAD Web App*. 2019. Available also from: https://madewithwebassembly.com/showcase/autocad/. Accessed: 2025-12-28.
6. JORDON MEARS. *How we're bringing Google Earth to the web*. [N.d.]. Available also from: https://web.dev/case-studies/earth-webassembly/. Accessed: 2025-12-28.
7. ROSSBERG, Andreas. *WebAssembly Core Specification*. 2019-12-05. W3C. Available also from: https://www.w3.org/TR/wasm-core-1/.

Contents of the attachment

```
/
├── readme.txt ........................... stručný popis obsahu média
├── exe .................... adresář se spustitelnou formou implementace
├── src
│   ├── impl ............................. zdrojové kódy implementace
│   └── thesis .................. zdrojová forma práce ve formátu LaTeX
└── text ................................................. text práce
    └── thesis.pdf .......................... text práce ve formátu PDF
```