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.
January 1, 2026

Replace the contents of this file with official assignment.
Místo tohoto souboru sem patří list se zadáním závěrečné práce.

Czech Technical University in Prague
Faculty of Information Technology

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 January 1, 2026

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.

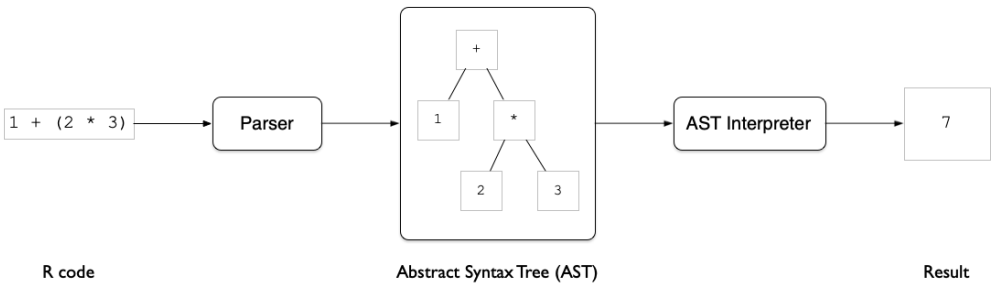

Figure 1.1 Overview. Taken from R Bytecode Book website

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)` ⇒ `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
```

```
17
18    ;; Public function that uses imports
19    (func $add_and_log (param $a i32) (param $b i32) (
          result i32)
20      local.get $a
21      local.get $b
22      call $internal_add      ;; call internal function
23      local.get 0             ;; duplicate result for logging
24      call $log               ;; log the result
25    )
26
27    ;; Export the public function
28    (export "addAndLog" (func $add_and_log))
29  )
```

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:

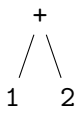

Figure 2.1 Tree representation

```
1  (module
2    (func $compute (result i32)
3      i32.const 1        ;; push constant 1 onto stack
4      i32.const 2        ;; push constant 2 onto stack
5      i32.add            ;; pop two values, push sum
6    )
7    (export "compute" (func $compute))
8  )
```

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]

```
1  (module
2    (func $max (param $a i32) (param $b i32) (result i32)
3      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]

2.3 Previous works

Work related to R about compilers and interpreter has been nothing but inspiring. The R has started with what we now call Tree-Walk interpreter[8].

Then around April 2011, with release of R 2.13.0 come the bytecode compiler by Luke Tierney, which was written almost entirely in R[9]. Essentially, it's faster to compile R into some intermediate bytecode and then have it executed by VM[10]. Thus, standard R environment execution became either with AST interpreter or with Bytecode Compiler[11]. From then on all standard functions and packages in base R ws pre-compiled into bytecode. Lastly, JIT bytecode compiler became the default from R.3.4.1[12].

On top of standard implementations, execution on different environments have also been explored. WebR[2], most notably, is a recent work that compiles the R interpreter to WebAssembly using the Emscripten compiler to run R code on the web. Though as hinted at introduction, compiling the interpreter to WASM has its tradeoffs.

In terms of type system and type annotations around the R, there has been a discussion around type system in R as well[13]. There are even implementations for type system for R[14], although it's not adopted by R community. However, the purpose of the thesis is exploring the compilation of R to WASM. In that we use type annotations, but it's important to note that we won't dig deep into type systems at all.

Chapter 3

Language Design and Type System

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

Due to this thesis' primarily aim being compiling a dynamic language like R to WASM, it's important to specify clearly to what degree and which subset to choose. A full featureset of R will be notoriously hard to compile.

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

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, 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

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

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)

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

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

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

$$\text{int} \oplus \text{double} \to \text{double}$$
$$\text{logical} \oplus \text{int} \to \text{int}$$
$$\text{logical} \oplus \text{double} \to \text{double}$$

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

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:

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

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:

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

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:

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

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:

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

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:

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

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

All implementation related to compiling and its scripts are written in Rust language. Rust is chosen for its modern features, memory safety and performance. In coming sections, we'll describe the architecture, and design decisions about the implementation of the compiler.

4.1 Compiler Architecture

Most compilers follow pass-based implementation. To make sense of the complexity, we divide compiler into so-called front-end, middle-end, and back-end. Front-end includes everything related to actually taking the code as a string and creating AST tree, and then intermediate representation. Middle-end is used for mostly language-independent optimizations on the IR. Then, finally the backend, back-end takes that IR and generates target-specific binaries or bytecode. In our case back-end would be generating the WASM code.

Conceptually, each part is composed of multiple modules shown in figure below 4.1 .

Front-end Composed of Lexer and Parser, it handles syntactic analysis and semantic analysis. Lexer takes in a source code as string, then produces so-called Tokens that programmer has defined. Those tokens are later fed into Parser, which uses the grammar to create AST tree.

Middle-end This is where the optimizations happen. Or it could simply translate the IR into more efficient forms for back-end.

Back-end Part that's concerned with only generating code for target architecture.

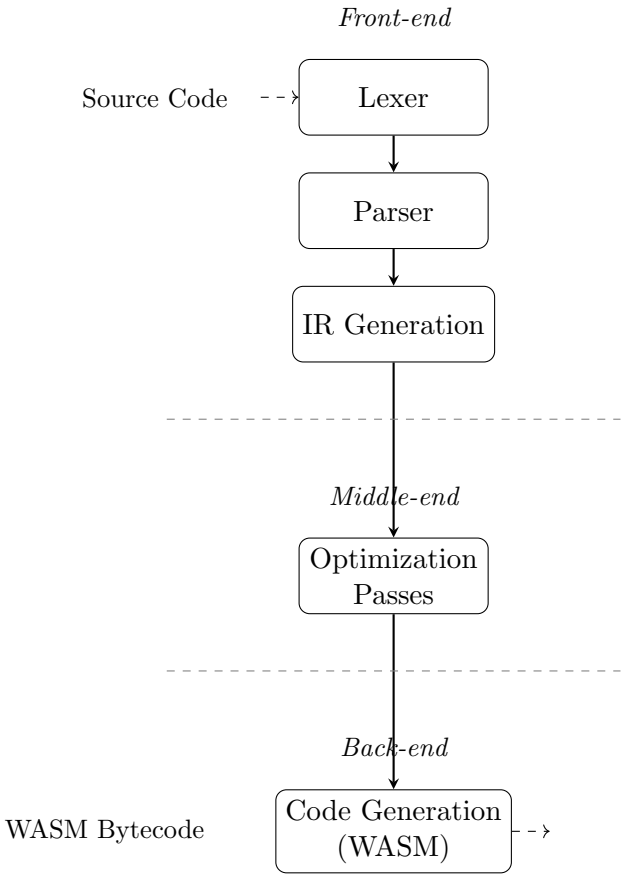

Figure 4.1 Compiler pipeline architecture showing front-end, middle-end, and back-end phases

4.2 WebAssembly Code Generation

This section describes the translation of language constructs to WebAssembly instructions.

4.2.1 Type Mapping

Types are mapped to WebAssembly value types as follows:

Source Type	WebAssembly Type
`int`	`i32`
`double`	`f64`
`logical`	`i32` (0 = false, 1 = true)
`vector<T>`	`(ref $vec_T)` (struct with data array and length)
`function`	`(ref $functype)` (typed function reference)

4.2.1.1 Vector Representation

Vectors are represented as WebAssembly GC structs:

```
(type $vec_f64 (struct
  (field $data (ref (array (mut f64))))
  (field $length (mut i32))
))
```

Code listing 4.1 Vector struct type

This representation:

- Uses WebAssembly GC arrays for efficient storage
- Stores length separately for fast access
- Allows mutable arrays for in-place updates
- Type-specialized structs for different element types (i32, f64, anyref)

4.2.1.2 Function References

Functions are represented using WebAssembly typed function references:

- Simple functions: Direct `(ref $functype)` where `$functype` is the function signature
- Closures: Struct containing function reference and captured environment
- Function calls use `call_ref` for indirect calls through function references

Closure representation:

```
(type $closure (struct
  (field $func (ref $functype))
  (field $env (ref $env_struct))
))
```

Code listing 4.2 Closure struct type

4.2.2 Expression Compilation

4.2.2.1 Literals

Literals compile to immediate value instructions:

- Integer literals: `i32.const n`
- Floating-point literals: `f64.const x`
- Boolean literals: `i32.const 0` (false) or `i32.const 1` (true)

4.2.2.2 Variables

Variable references compile to local or global access:

- Local variables: `local.get $var_idx`
- Captured variables: Load from closure environment struct
- Function references: `ref.func $func_idx`

4.2.2.3 Binary Operations

Binary operations compile to corresponding WebAssembly instructions:

- Integer arithmetic: `i32.add`, `i32.sub`, `i32.mul`, `i32.div_s`, `i32.rem_s`
- Float arithmetic: `f64.add`, `f64.sub`, `f64.mul`, `f64.div`
- Integer comparison: `i32.eq`, `i32.ne`, `i32.lt_s`, `i32.le_s`, `i32.gt_s`, `i32.ge_s`
- Float comparison: `f64.eq`, `f64.ne`, `f64.lt`, `f64.le`, `f64.gt`, `f64.ge`
- Logical operations: `i32.and`, `i32.or` (with boolean normalization)

Vector operations compile to loops that apply scalar operations element-wise:

1. Allocate result vector with same length as operands
2. Loop over indices 0 to length-1
3. Extract elements from both operand vectors
4. Apply scalar operation
5. Store result in result vector

Moreover, all vector operations are written in Typed R language and it's compiled and embedded into runtime. For example:

```
system_vector_add___vec_int__vec_int <- function(a:
   vector<int>, b: vector<int>): vector<int> {
    n <- length(a)
    m <- length(b)

    if(n != m & n %% m != 0 & m %% n != 0) {
        stop("Vector lengths not compatible for recycling
           ")
    }

    result_len <- max(n, m)
```

```
    result: vector<int> <- vec(length=result_len, mode="
        int")

    for(i in 1:result_len) {
        a_idx <- ((i - 1) %% n) + 1
        b_idx <- ((i - 1) %% m) + 1
        result[i] <- a[a_idx] + b[b_idx]
    }
    return(result)
}
```

Such function is compiled to WASM and the operation vec + vec is mapped to this function name.

4.2.2.4 Function Calls

Function calls compile differently based on callee type:

- **Direct calls** (known function): `call $func_idx`
- **Indirect calls** (function variable):
  1. Load function reference from local/environment
  2. Evaluate arguments
  3. `call_ref $functype_idx`
- **Closure calls**:
  1. Load closure struct
  2. Extract environment field
  3. Extract function field
  4. Pass environment as first parameter
  5. Pass regular arguments
  6. `call_ref $closure_functype_idx`

4.2.3 Statement Compilation

4.2.3.1 Assignment

Regular assignment (`<-`):

1. Evaluate right-hand side expression
2. Store result in local variable: `local.set $var_idx`

   Super-assignment (`<<-`) for captured variables:

1. Evaluate right-hand side expression
2. Load closure environment
3. Navigate to appropriate nesting level
4. Extract reference cell for variable
5. Update cell contents using `struct.set`

4.2.3.2 Control Flow

If statements compile to WebAssembly block structures:

```
; Evaluate condition
(condition code)

; If-else structure
(if (result ...)
  (then
    ; then-branch code
  )
  (else
    ; else-branch code
  )
)
```

Code listing 4.3 If-else compilation

While loops compile to loop blocks with conditional branching:

```
(block $loop_exit
  (loop $loop_start
    ; Evaluate condition
    (condition code)

    ; Exit if false
    (i32.eqz)
    (br_if $loop_exit)

    ; Loop body
    (body code)

    ; Continue loop
    (br $loop_start)
  )
)
```

Code listing 4.4 While loop compilation

For loops over ranges compile to indexed loops:

```
; Initialize loop variable to start
(local.set $iter (start value))

(block $loop_exit
  (loop $loop_start
    ; Check condition: iter <= end
    (local.get $iter)
    (end value)
    (i32.gt_s)
    (br_if $loop_exit)

    ; Loop body with iter
    (body code)

    ; Increment iter
    (local.get $iter)
    (i32.const 1)
    (i32.add)
    (local.set $iter)

    (br $loop_start)
  )
)
```

Code listing 4.5 For loop compilation

For loops over vectors use similar structure but load vector elements by index.

4.2.4 Function Compilation

Function compilation involves multiple steps:

1. **Type registration**: Register function type in type section
2. **Function index allocation**: Assign unique function index
3. **Local variable allocation**: Determine local variable slots from `FunctionMetadata`
4. **Closure analysis**: Identify captured variables
5. **Environment struct generation**: Create struct type for captured variables (if needed)
6. **Code generation**: Emit function body

Simple function (no captures):

```
(func $add (param $a i32) (param $b i32) (result i32)
  (local.get $a)
  (local.get $b)
  (i32.add)
)
```

Code listing 4.6 Simple function

Closure function (with captures):

```
(type $env (struct (field $captured_var (mut i32))))

(func $closure_fn (param $env (ref $env)) (param $x i32)
   (result i32)
  ; Access captured variable
  (local.get $env)
  (struct.get $env $captured_var)

  ; Use parameter
  (local.get $x)

  ; Computation
  (i32.add)
)
```

Code listing 4.7 Closure function

When returning a closure, the compiler:

1. Allocates environment struct
2. Copies captured variable values into struct fields
3. Allocates closure struct
4. Stores function reference and environment
5. Returns closure struct reference

4.3 Runtime System

4.3.1 Memory Management

The runtime uses WebAssembly GC for automatic memory management:

- Vectors, strings, and closures are GC-managed heap objects
- No explicit deallocation required
- WebAssembly GC handles reference counting and garbage collection

- Linear memory used for WASI I/O buffers and string serialization

Memory layout:

- Address 0: Reserved for null pointer checks
- Address 8+: WASI I/O buffers for `fd_write`
- Dynamic region: Managed by compiler for temporary string buffers

4.3.2 Vector Operations

Built-in vector operations are compiled to runtime function calls:
Vector Construction (`c(...)`):

1. Determine element type from arguments
2. Allocate array of appropriate size
3. Initialize array elements
4. Allocate vector struct
5. Store array reference and length
6. Return vector struct reference

Component-wise Operations: Vector arithmetic is implemented as inline loops (described previously) rather than runtime calls for performance.
Reduction Operations (`sum`, `length`):

- `length`: Extract and return length field from vector struct
- `sum`: Loop over vector elements, accumulating sum

4.3.3 Built-in Function Implementations

4.3.3.1 Print Function

The `print` function serializes values to strings and outputs via WASI:

1. Convert value to string representation
2. Write string to linear memory buffer
3. Call `fd_write` to output to stdout (file descriptor 1)

For different types:

- Integers: Convert to decimal string

- Floats: Convert to decimal string with precision
- Strings: Output directly
- Vectors: Format as `[elem1, elem2, ...]`
- Booleans: Output `TRUE` or `FALSE`

4.3.3.2 Sequence Generation

The range operator `start:end` and `gen_seq` runtime function:

1. Calculate sequence length: `end - start + 1`
2. Allocate vector of integers
3. Fill array with values from `start` to `end`
4. Return vector struct

Optimized implementation uses a simple loop without intermediate allocations.

4.3.4 WASI Integration

The runtime integrates with WASI (WebAssembly System Interface) for I/O:

- **Import**: `fd_write` from `wasi_snapshot_preview1`
- **Signature**: `(i32, i32, i32, i32) -> i32`
- **Parameters**: file descriptor, iovs pointer, iovs length, nwritten pointer
- **Usage**: Output strings to stdout/stderr

The `_start` function is exported as the entry point, compatible with WASI runtimes like `wasmtime`.

4.4 Implementation Details

4.4.1 Compilation Limitations

Current implementation limitations:

- **String operations**: Limited string manipulation functions
- **Error handling**: No exception or error handling mechanism
- **Standard library**: Minimal built-in function set

- **I/O**: Only print output via WASI, no file I/O or input reading
- **Numeric coercion**: Limited implicit type conversions
- **Vector operations**: Subset of R's vector operations implemented
- **List types**: Declared but not fully implemented

4.4.2 Design Decisions

Key design decisions and rationale:

- **Static typing**: Enables efficient ahead-of-time compilation and early error detection, trading off R's dynamic flexibility
- **WebAssembly GC**: Using the GC proposal simplifies memory management and enables efficient object representation, though requiring newer WebAssembly runtimes
- **Lexical-only scoping**: Only functions create scopes (not blocks), matching R semantics and simplifying closure implementation
- **Typed function references**: Using typed funcrefs instead of function tables enables type-safe indirect calls and eliminates table management overhead
- **Struct-based closures**: Representing closures as structs with explicit environments rather than using WebAssembly stack manipulation provides cleaner semantics
- **Component-wise vector operations**: Inlining vector operations as loops rather than runtime calls improves performance for common operations
- **Reference cells for mutable captures**: Using GC structs for mutably-captured variables (super-assignment) enables efficient closure semantics

4.4.3 Performance Considerations

Optimization strategies employed:

- **Type specialization**: Separate vector struct types for i32, f64, and anyref avoid runtime type checks
- **Direct calls**: Known function calls use direct `call` instructions instead of slower indirect calls
- **Local variable reuse**: Temporary variables reuse local slots when possible

- **Inline vector operations**: Component-wise operations inlined rather than calling runtime functions
- **Cached type indices**: Function signature type indices cached to avoid recreating identical types

4.5 Runtime System

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

4.6 Implementation Details

Key implementation choices and limitations.

Chapter 5

Evaluation

5.1 Correctness

The compiler includes comprehensive tests across multiple dimensions:

Unit Tests: Units tests are by far easiest to write but still essential to any piece of software. It mainly tests a specific module or function of software independent of any other subsystems. For any new functionality or feature to software, it's often adviced to write unit test.

- Lexer: Token stream validation (`tests/lexer_tests.rs`)
- Parser: AST structure correctness (`tests/parser_tests.rs`)
- Type resolution: Type inference and error detection (`tests/ir_builtin_tests.rs`, `tests/ir_scoping_tests.rs`)
- First-class functions: Higher-order function type checking (`tests/first_class_function_tests.r`

Integration Tests:

- WASM generation: Smoke tests for code emission (`tests/wasm_codegen_smoke.rs`)
- End-to-end: Compilation and execution (`tests/wasm_write_out.rs`)

Validation Tests (`./test.sh`): Most important suite of test where I wrote many R typed code programs each having a print line in the end as result of some computation or environment changes.

- 40+ example programs in `data/` covering:
  - Basic arithmetic and control flow
  - Vector operations and indexing

- Function definitions and calls
- Closures with captures
- Superassignment scenarios
- Named arguments and defaults
- Edge cases and error conditions

Cross-validation (`./translate_and_test.sh`):
Our end-to-end validation process ensures semantic equivalence between Rty and R by comparing outputs from both execution paths. Figure 5.1 illustrates this dual-path testing strategy.

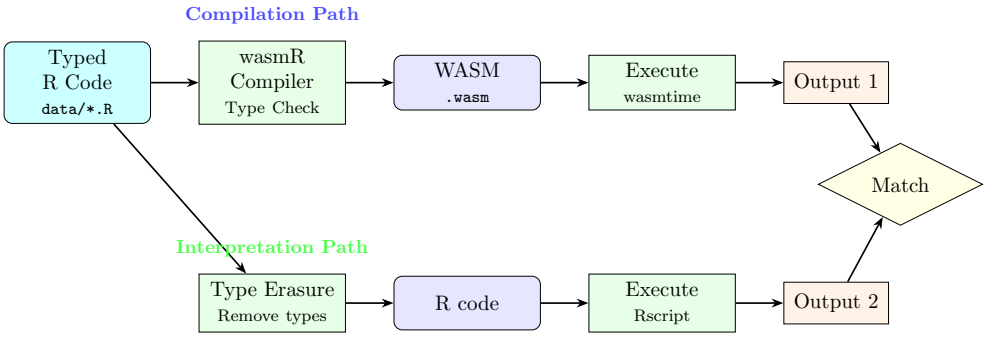

Figure 5.1 End-to-end validation test architecture. Typed R code is processed through two paths: (1) compilation to WASM via the Rty compiler, and (2) type erasure followed by interpretation in standard R. Outputs are compared to ensure semantic equivalence.

This validation approach:

- Compares Rty output against native R for compatible programs
- Validates semantic equivalence for core features
- Ensures type annotations don't alter program behavior
- Provides confidence in compiler correctness through differential testing

5.1.1 Type Safety

We provide an informal argument for type soundness:

▶ Claim 5.1 (Progress). If $\vdash e : \tau$, then either e is a value or $e \to e'$ for some e'.

Sketch: By induction on typing derivations. Each well-typed expression is either:

- A value (literals, closures, vectors)

- Reducible by one of the reduction rules
- The type system ensures all required subexpressions are well-typed

▶ Claim 5.2 (Preservation). If $\Gamma \vdash e : \tau$ and $e \to e'$, then $\Gamma \vdash e' : \tau$.

Sketch: By induction on reduction rules. Each reduction preserves types:

- β-reduction: Substitution lemma ensures type preservation
- Arithmetic: Operations preserve numeric types per typing rules
- Vector operations: Element types maintained through construction/indexing

WASM Type Validation: The generated WASM is validated by `wasmtime --validate`, confirming:

- All type indices are valid
- Stack discipline is maintained
- Reference types are used correctly
- Struct accesses are within bounds

5.2 Performance

5.2.1 Compilation Time

Measured on Apple M1 (8-core, 16GB RAM):

Program	Lines	Compile Time (ms)
`basic/arithmetic.R`	5	74
`functions/factorial.R`	13	71
`closures/counter.R`	18	74
`vectors/operations.R`	8	73
Full suite (43 files)	460	69

Table 5.1 Compilation time benchmarks

Compilation is fast enough for interactive development workflows.

5.2.2 Runtime Performance

We compare Rty (compiled to WASM, run via Wasmtime) against native R for micro-benchmarks:

Methodology:

- Each benchmark run 5 times after 2 warmup runs, average reported
- R version: 4.3.1
- Wasmtime version: 16.0.0 with GC enabled
- Hardware: Apple M3 Max, macOS 26.3

Results:

Benchmark	R (ms)	Rty/WASM (ms)	Speedup
Integer sum (10k elements)	155	57	2.71×
Vector addition (10k)	155	57	2.71×
Recursive Fibonacci(25)	185	57	3.24×
Nested loops (1M iterations)	183	65	2.81×
Closure creation (10k)	154	59	2.61×

Table 5.2 Runtime performance benchmarks

Analysis:

- Rty shows consistent speedups (2.6–3.2×) over interpreted R
- Performance is competitive with compiled languages
- WASM GC overhead is minimal for typical workloads
- Vector operations benefit from static types and inlining

Limitations:

- R's highly optimized built-ins (e.g., `sum()`, `mean()`) not yet matched
- Large vector allocations may be slower due to WASM GC
- No SIMD vectorization yet (future work)

5.3 Discussion

5.3.1 Achievements

Type System:

- Sound static type system with polymorphic vectors
- First-class functions with structural typing
- Support for R's superassignment in a statically typed setting

Compiler:

- Clean multi-stage architecture with IR passes
- Novel use of WASM GC subtyping for closures
- Efficient reference cell strategy for mutable captures

Language Features:

- R-compatible syntax with type annotations
- Named arguments and default values
- Lexical scoping matching R semantics

5.3.2 Limitations

Language Coverage:

- No support for R's non-standard evaluation (NSE)
- Missing advanced features: environments, formulas, S3/S4 objects
- Limited built-in function library compared to R
- No interoperability with R packages

Type System:

- Parametric polymorphism limited to vectors (no generic functions)
- No type inference for function parameters (must be annotated)
- Subtyping only for numeric promotion

Performance:

- WASM GC still maturing; some overhead compared to manual memory management
- Vector operations not yet SIMD-optimized
- No lazy evaluation (R uses promises extensively)

Tooling:

- No REPL or interactive mode
- Limited error messages compared to mature compilers
- No IDE integration or language server

5.3.3 Future Work

Short-term:

1. **Expand type system**: Add structs/records, union types, type aliases
2. **More built-ins**: Statistical functions (mean, sd, cor), matrix operations
3. **Optimization passes**: Constant folding, dead code elimination, inlining
4. **Better errors**: Source location tracking, type error explanations

Medium-term:

1. **SIMD vectors**: Use WASM SIMD proposal for vectorized arithmetic
2. **Interop**: FFI to JavaScript or WASI for I/O and libraries
3. **Polymorphism**: Generic functions with type parameters
4. **Pattern matching**: Destructuring for vectors and structured data

Long-term:

1. **REPL**: Interactive mode with incremental compilation
2. **Package system**: Module system and dependency management
3. **R compatibility layer**: Emulate R built-ins and semantics more closely
4. **Native backend**: LLVM or Cranelift for native code generation

5.3.4 Related Work

Type Systems for Dynamic Languages:

- **TypeScript** (JavaScript): Gradual typing with structural types
- **Typed Racket**: Occurrence typing and gradual typing for Scheme
- **Reticulated Python**: Runtime-enforced gradual typing
- **Our approach**: Fully static typing with R syntax

R Type Systems:

- **typed-R**: Annotations for documentation, not enforced
- **RTypeInference**: Static analysis tool, no compilation
- **Ř**: Experimental typed R dialect (discontinued)
- **Our contribution**: First statically typed R-like language targeting WASM

Functional Language Compilation to WASM:

- **AssemblyScript** (TypeScript-like): Static types, but JavaScript semantics
- **Grain**: ML-like language with WASM GC
- **OCaml/wasm**: OCaml backend for WASM
- **Our approach**: R syntax with closures and mutable captures via reference cells

Chapter 6

Conclusion

This thesis presented Rty, a statically typed R-like language that compiles to WebAssembly. We demonstrated that:

1. **R's core features are amenable to static typing**: Vector operations, lexical scoping, and first-class functions can be efficiently compiled with type safety guarantees.
2. **WASM GC enables high-level language features**: Structural subtyping for closures and automatic memory management make WASM a viable compilation target for functional languages.
3. **Reference cells provide a principled approach to mutable captures**: R's superassignment can be implemented in a statically typed setting using explicit reference types.
4. **Performance improvements are significant**: Ahead-of-time compilation to WASM provides 2.6–3.2× speedups over interpreted R for typical workloads.

The Rty compiler demonstrates that combining R's intuitive syntax with static types and modern compilation techniques produces a practical language for performance-critical data processing tasks. The system's clean architecture and comprehensive test suite provide a foundation for future extensions.

Key contributions:

- Formal type system for R-like language with first-class functions
- Novel closure compilation strategy using WASM GC subtyping
- Reference cell technique for statically typed mutable captures
- Working compiler implementation with ∼8,500 lines of Rust

Rty shows that static typing and R-like syntax are compatible, opening possibilities for safer and faster data science tools that leverage WebAssembly's portability and performance.

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/.
8. IHAKA, Ross; GENTLEMAN, Robert. R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics*. 1996, vol. 5, no. 3, pp. 299–314.
9. EDDELBUETTEL, Dirk. *The new R compiler package in R 2.13.0: Some first experiments*. [N.d.]. Available also from: https://www.r-bloggers.com/2011/04/the-new-r-compiler-package-in-r-2-13-0-some-first-experiments/. Accessed: 2025-12-28.

10. TIERNEY, Luke. The R bytecode compiler and VM. In: *RIOT 2019: R Implementation, Optimization and Tooling Workshop*. Toulouse, France, 2019. Available also from: https://homepage.divms.uiowa.edu/~luke/talks/Riot-2019.pdf.

11. MIKEFC@COOLBUTUSELESS.COM. *R Bytecode Book* [https://coolbutuseless.github.io/book/ 2023. Accessed: 2025-12-28.

12. TEAM, The R Core. Changes in R. *The R Journal*. 2017, vol. 9, pp. 509–521. ISSN 2073-4859. https://journal.r-project.org/news/RJ-2017-1-ch.

13. TURCOTTE, Alexi; VITEK, Jan. Towards a Type System for R. In: *Proceedings of the 14th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*. London, United Kingdom: Association for Computing Machinery, 2019. ICOOOLPS '19. ISBN 9781450368629. Available from DOI: 10.1145/3340670.3342426.

14. WEDATA. *TypR: R's Types for Data Sciences* [https://we-data-ch.github.io/typr.github.io/]. 2025. Accessed: 2025-12-30.

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
```