



# COMPILING A SUBSET OF CLOJURE WITH LLVM

**Jan Šuráň**

Bachelor's thesis  
Faculty of Information Technology  
Czech Technical University in Prague  
Department of theoretical computer science  
Study program: Informatics  
Specialization: Theoretical computer science  
Supervisor: Pierre Donat-Bouillud, Ph.D.  
May 15, 2026



## Assignment of bachelor's thesis

**Title:** Compiling a subset of Clojure with LLVM  
**Student:** Jan Šuráň  
**Supervisor:** Pierre Donat-Bouillud, Ph.D.  
**Study program:** Informatics  
**Branch /** Computer Science 2021  
**specialization:** Department of Theoretical Computer Science  
**Department:** until the end of summer semester 2027/2028  
**Validity:**

### Instructions

Clojure is a modern dialect of Lisp that runs on the JVM and supports dynamic typing, immutable data structures, first-class functions, and a macro system. While Clojure benefits from the JVM ecosystem, it is tightly coupled to its runtime and compilation model. LLVM is a modular compiler framework that enables native code generation for multiple architectures through its intermediate representation (LLVM IR). It is widely used to build optimizing compilers for modern languages.

The goal of this thesis is to define and implement a dynamically-typed subset of Clojure and compile it directly to LLVM IR. The subset should be small enough to be implementable within a bachelor thesis, yet expressive enough to support core Lisp features.

1. Research Clojure and LLVM
2. Define a subset of Clojure. The subset should include numbers, strings, lists, first-class functions, simple control flow (conditionals and recursion), and a simplified macro system
3. Implement a compiler from this subset to LLVM IR
4. Design a runtime system: implement memory management and runtime support for dynamic typing, lists, and closures.
5. Evaluate the system: assess correctness, expressiveness, and performance compared to JVM-based Clojure.

---

*Electronically approved by doc. RNDr. Dušan Knop, Ph.D.  
on 10 March 2026 in Prague.*



**FACULTY  
OF INFORMATION  
TECHNOLOGY  
CTU IN PRAGUE**

- [1] Clojure documentation: <https://clojure.org>
- [2] LLVM documentation: <https://llvm.org>
- [3] Queinnec, C. Lisp in Small Pieces
- [4] Appel, A. W. Modern Compiler Implementation
- [5] <https://bernsteinbear.com/blog/compiling-a-lisp-0/>



---

*Electronically approved by doc. RNDr. Dušan Knop, Ph.D.  
on 10 March 2026 in Prague.*

Czech Technical University in Prague

Faculty of Information Technology

© 2026 Jan Šurán. All rights reserved.

*This thesis is school work as defined by Copyright Act of the Czech Republic.*

*It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).*

Citation of this thesis: Jan Šurán. *Compiling a subset of Clojure with LLVM*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2026.

*I would like to thank my supervisor, Pierre Donat-Bouillud, Ph.D., for his guidance and valuable feedback during my work on this thesis. I would like to thank my friends and my family for the support during my studies and their positive energy to push me forward.*

## Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I declare that I have used AI tools during the preparation and writing of my thesis. I have verified the generated content. I confirm that I am aware that I am fully responsible for the content of the thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Czech Technical University in Prague has the right to conclude a licence agreement on the utilization of this thesis as a school work pursuant of Section 60(1) of the Act.

In Praha on May 15, 2026

## Abstract

This thesis presents the design and implementation of TinyClojure, a compiler for a restricted subset of the Clojure programming language targeting LLVM IR. Clojure is a modern Lisp dialect most commonly implemented on the Java Virtual Machine, which provides a mature runtime environment but also tightly couples the language implementation to JVM services such as class loading, garbage collection, and dynamic dispatch.

The goal of this work is to explore how a small dynamically typed Lisp-like language can be compiled directly to LLVM IR. TinyClojure supports core language features such as literals, lists, symbols, Vars, first-class functions, closures, conditionals, recursion, and a simplified macro system. The implementation consists of a reader, macroexpansion and semantic analysis phases, LLVM IR generation, and a supporting runtime system implemented in C++.

Because LLVM does not provide high-level language runtime services, the thesis also describes the design of TinyClojure's runtime representation of dynamic values, function objects, closures, global Vars, module loading, and memory management. The compiler supports both just-in-time execution and ahead-of-time compilation of modules that can be loaded at runtime.

The implementation is evaluated in terms of correctness, expressiveness, and performance. The results show that the implemented subset can be compiled and executed using LLVM, and illustrate the main design trade-offs involved in implementing a dynamically typed Lisp-like language without relying on the JVM runtime.

**Keywords** Clojure, Lisp, TinyClojure, compilers, LLVM IR, just-in-time compilation, ahead-of-time compilation, dynamic typing, C++

## Abstrakt

Tato práce představuje návrh a implementaci TinyClojure, překladače omezené podmnožiny programovacího jazyka Clojure do LLVM IR. Clojure je moderní dialekt Lispu, který je nejčastěji implementován nad platformou Java Virtual Machine. Ta poskytuje vyspělé běhové prostředí, ale zároveň úzce váže implementaci jazyka na služby JVM, jako je načítání tříd, garbage collection a dynamické dispatchování.

Cílem této práce je prozkoumat, jak lze malý dynamicky typovaný jazyk podobný Lispu překládat přímo do LLVM IR. TinyClojure podporuje základní jazykové prvky, jako jsou literály, seznamy, symboly, Vars, funkce první třídy, uzávěry, podmínky, rekurze a zjednodušený systém maker. Implementace se skládá z readeru, fází makroexpandy a sémantické analýzy, generování LLVM IR a podpůrného běhového prostředí implementovaného v C++.

Protože LLVM neposkytuje vysokoúrovňové služby běhového prostředí, práce dále popisuje návrh běhové reprezentace dynamických hodnot, funkčních objektů, uzávěrů, globálních Vars, načítání modulů a správy paměti v TinyClojure. Překladač podporuje jak just-in-time spouštění, tak ahead-of-time překlad modulů, které lze následně načítat za běhu programu.

Implementace je vyhodnocena z hlediska korektnosti, expresivity a výkonu. Výsledky ukazují, že implementovanou podmnožinu lze překládat a spouštět pomocí LLVM, a ilustrují hlavní návrhové kompromisy spojené s implementací dynamicky typovaného jazyka podobného Lispu bez využití běhového prostředí JVM.

**Klíčová slova** Clojure, Lisp, TinyClojure, překladače, LLVM IR, just-in-time překlad, ahead-of-time překlad, dynamické typování, C++

## Contents

|                                                             |    |
|-------------------------------------------------------------|----|
| Introduction .....                                          | 1  |
| 1 Background .....                                          | 3  |
| 1.1 Lisp and Clojure .....                                  | 3  |
| 1.2 Source language model .....                             | 3  |
| 1.3 Standard architecture of modern compilers .....         | 4  |
| 1.3.1 Lexical analysis and reading .....                    | 4  |
| 1.3.2 Syntactic analysis .....                              | 5  |
| 1.3.3 Semantic analysis .....                               | 5  |
| 1.3.4 Optimization .....                                    | 6  |
| 1.3.5 Code generation .....                                 | 6  |
| 1.4 Java and the JVM .....                                  | 6  |
| 1.4.1 Execution model .....                                 | 7  |
| 1.4.2 Class loading and initialization .....                | 8  |
| 1.5 JVM Clojure .....                                       | 8  |
| 1.5.1 Namespaces and Vars .....                             | 9  |
| 1.5.2 Function invocation .....                             | 10 |
| 1.5.3 Closures .....                                        | 11 |
| 1.5.4 Compilation model .....                               | 11 |
| 1.6 Jank .....                                              | 12 |
| 1.7 LLVM .....                                              | 13 |
| 1.7.1 LLVM IR .....                                         | 13 |
| 1.7.2 SSA and phi nodes .....                               | 14 |
| 1.7.3 LLVM modules .....                                    | 15 |
| 1.7.4 LLVM C++ API .....                                    | 16 |
| 1.8 Other native implementations of dynamic languages ..... | 18 |
| 2 The TinyClojure specification .....                       | 19 |
| 2.1 Reader syntax .....                                     | 19 |
| 2.1.1 Whitespace and comments .....                         | 20 |
| 2.1.2 Integer literals .....                                | 20 |
| 2.1.3 Floating point literals .....                         | 20 |
| 2.1.4 Character literals .....                              | 20 |
| 2.1.5 String literals .....                                 | 21 |
| 2.1.6 Symbolic value literals .....                         | 21 |
| 2.1.7 Symbols .....                                         | 22 |
| 2.1.8 Lists .....                                           | 22 |
| 2.1.9 Quote .....                                           | 22 |
| 2.1.10 Syntax quote .....                                   | 22 |
| 2.2 Runtime values .....                                    | 24 |

|       |                                                      |    |
|-------|------------------------------------------------------|----|
| 2.2.1 | Nil and booleans .....                               | 24 |
| 2.2.2 | Numbers .....                                        | 25 |
| 2.2.3 | Lists .....                                          | 25 |
| 2.2.4 | Vars .....                                           | 25 |
| 2.2.5 | Functions and closures .....                         | 25 |
| 2.3   | Evaluation model .....                               | 26 |
| 2.3.1 | Self-evaluating forms .....                          | 26 |
| 2.3.2 | Symbol resolution .....                              | 26 |
| 2.3.3 | List evaluation .....                                | 26 |
| 2.3.4 | Function application .....                           | 26 |
| 2.4   | Special forms .....                                  | 26 |
| 2.4.1 | <code>quote</code> .....                             | 27 |
| 2.4.2 | <code>if</code> .....                                | 27 |
| 2.4.3 | <code>do</code> .....                                | 27 |
| 2.4.4 | <code>def</code> .....                               | 28 |
| 2.4.5 | <code>let*</code> .....                              | 28 |
| 2.4.6 | <code>recur</code> .....                             | 29 |
| 2.4.7 | <code>loop*</code> .....                             | 29 |
| 2.4.8 | <code>var</code> .....                               | 29 |
| 2.4.9 | <code>fn*</code> .....                               | 30 |
| 2.5   | Functions .....                                      | 30 |
| 2.5.1 | Basic function syntax .....                          | 30 |
| 2.5.2 | Multiple arities .....                               | 31 |
| 2.5.3 | Variadic functions .....                             | 32 |
| 2.5.4 | Closures .....                                       | 33 |
| 2.5.5 | Function invocation .....                            | 33 |
| 2.6   | Macros .....                                         | 34 |
| 2.7   | Modules and top-level forms .....                    | 34 |
| 2.8   | Errors and limitations .....                         | 35 |
| 2.9   | Core library .....                                   | 36 |
| 3     | Runtime system .....                                 | 38 |
| 3.1   | Object representation .....                          | 39 |
| 3.1.1 | Method tables .....                                  | 39 |
| 3.1.2 | Runtime helper functions .....                       | 41 |
| 3.1.3 | Why the method table is stored in each object .....  | 42 |
| 3.2   | Runtime value types .....                            | 42 |
| 3.3   | Function representation and calling convention ..... | 43 |
| 3.3.1 | Uniform calling convention .....                     | 44 |
| 3.3.2 | Arity dispatch .....                                 | 44 |
| 3.3.3 | Built-in functions .....                             | 45 |
| 3.3.4 | Closures .....                                       | 46 |

|       |                                                 |    |
|-------|-------------------------------------------------|----|
| 3.3.5 | Dynamic invocation and direct linking .....     | 46 |
| 3.4   | Vars and global environment .....               | 47 |
| 3.4.1 | Vars .....                                      | 48 |
| 3.4.2 | Global environment .....                        | 48 |
| 3.4.3 | Top-level definitions .....                     | 48 |
| 3.4.4 | Macros and Vars .....                           | 49 |
| 3.5   | Module loading support .....                    | 49 |
| 3.5.1 | Runtime module state .....                      | 49 |
| 3.5.2 | Loading dependencies .....                      | 50 |
| 3.5.3 | Executing top-level code .....                  | 50 |
| 3.5.4 | Running and loading modules .....               | 51 |
| 3.6   | Memory management .....                         | 52 |
| 3.6.1 | Allocation .....                                | 52 |
| 3.6.2 | Root set .....                                  | 53 |
| 3.6.3 | Collection points .....                         | 55 |
| 3.6.4 | Mark phase .....                                | 55 |
| 3.6.5 | Sweep phase .....                               | 56 |
| 3.6.6 | Limitations .....                               | 57 |
| 3.6.7 | Alternative garbage-collection approaches ..... | 57 |
| 4     | Compiler implementation .....                   | 59 |
| 4.1   | Reader .....                                    | 60 |
| 4.1.1 | Syntax quote lowering .....                     | 60 |
| 4.2   | Macroexpansion .....                            | 61 |
| 4.3   | Semantic analysis .....                         | 62 |
| 4.3.1 | Symbol resolution .....                         | 63 |
| 4.3.2 | Lexical scopes and captured variables .....     | 63 |
| 4.3.3 | Special forms .....                             | 64 |
| 4.3.4 | Function analysis and closures .....            | 64 |
| 4.3.5 | Recur analysis .....                            | 65 |
| 4.4   | Function compilation .....                      | 65 |
| 4.4.1 | Function modules .....                          | 66 |
| 4.4.2 | Arity dispatch .....                            | 66 |
| 4.4.3 | Parameters and local bindings .....             | 67 |
| 4.4.4 | Captured values .....                           | 67 |
| 4.5   | LLVM IR generation .....                        | 68 |
| 4.5.1 | Emission model .....                            | 68 |
| 4.5.2 | Expression evaluation in the REPL .....         | 69 |
| 4.5.3 | Literals and runtime values .....               | 69 |
| 4.5.4 | Local bindings .....                            | 70 |
| 4.5.5 | Vars .....                                      | 72 |
| 4.5.6 | Function calls .....                            | 73 |

|        |                                                        |     |
|--------|--------------------------------------------------------|-----|
| 4.5.7  | Direct linking .....                                   | 74  |
| 4.5.8  | Built-in functions .....                               | 75  |
| 4.5.9  | do blocks .....                                        | 77  |
| 4.5.10 | Conditionals .....                                     | 77  |
| 4.5.11 | Loops and recur .....                                  | 78  |
| 4.5.12 | Runtime helper calls and error paths .....             | 80  |
| 5      | Evaluation .....                                       | 82  |
| 5.1    | Correctness testing .....                              | 82  |
| 5.1.1  | Reader unit tests .....                                | 82  |
| 5.1.2  | End-to-end comparison with JVM Clojure .....           | 83  |
| 5.1.3  | Expected-failure tests .....                           | 84  |
| 5.1.4  | AOT compilation and loading tests .....                | 85  |
| 5.2    | Performance .....                                      | 87  |
| 5.2.1  | Benchmark methodology .....                            | 87  |
| 5.2.2  | Overall benchmark results .....                        | 89  |
| 5.2.3  | Effect of integer caching .....                        | 90  |
| 5.2.4  | Effects of LLVM optimizations and direct linking ..... | 91  |
| 5.2.5  | Discussion .....                                       | 92  |
| 5.3    | Expressiveness and limitations .....                   | 93  |
| 5.4    | Future work .....                                      | 94  |
| 5.4.1  | Possible optimizations .....                           | 94  |
| 5.4.2  | Adding more features .....                             | 96  |
|        | Conclusion .....                                       | 99  |
|        | Bibliography .....                                     | 101 |
|        | Official KOS documents .....                           | 103 |
|        | Contents of the attachment .....                       | 107 |

## List of code listings

|            |                                                                                                                                                                               |    |
|------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----|
| Listing 1  | C program used as a running LLVM example. ....                                                                                                                                | 13 |
| Listing 2  | Simplified LLVM IR module for a C program that calls<br><code>printf</code> . ....                                                                                            | 14 |
| Listing 3  | Example usage of phi nodes in LLVM IR ....                                                                                                                                    | 15 |
| Listing 4  | Constructing the module from Listing 2 using the LLVM C++<br>API. ....                                                                                                        | 16 |
| Listing 5  | Simplified TinyClojure runtime object representation. ....                                                                                                                    | 39 |
| Listing 6  | Structure of the TinyClojure method table. ....                                                                                                                               | 40 |
| Listing 7  | Simplified runtime helper for TinyClojure equality. ....                                                                                                                      | 41 |
| Listing 8  | Schematic arity dispatch for fixed-arity overloads. ....                                                                                                                      | 45 |
| Listing 9  | Schematic arity dispatch with a variadic overload. ....                                                                                                                       | 45 |
| Listing 10 | Schematic representation of a closure object. ....                                                                                                                            | 46 |
| Listing 11 | Example of TinyClojure roots and reachability after a sequence<br>of top-level forms. The diagram shows the state after the final<br><code>def</code> form has finished. .... | 54 |
| Listing 12 | Schematic evaluation strategy used by the TinyClojure<br>REPL. ....                                                                                                           | 69 |
| Listing 13 | Schematic LLVM IR for constructing numeric literals. ....                                                                                                                     | 70 |
| Listing 14 | Schematic LLVM IR for constructing string and symbol<br>literals. ....                                                                                                        | 70 |
| Listing 15 | Schematic LLVM IR for a generated function method table. . .                                                                                                                  | 74 |
| Listing 16 | Schematic LLVM IR for a direct-linked function call. ....                                                                                                                     | 75 |
| Listing 17 | Schematic LLVM IR for a compiler-emitted arity error path. .                                                                                                                  | 81 |
| Listing 18 | AOT macroexpansion test module. The macro prints a message<br>when it is expanded and then returns its argument<br>unchanged. ....                                            | 86 |

## List of tables

|         |                                                                                      |    |
|---------|--------------------------------------------------------------------------------------|----|
| Table 1 | Examples of runtime objects, their payloads, and their static<br>method tables. .... | 40 |
| Table 2 | Main semantic-analysis steps performed by TinyClojure. ....                          | 62 |
| Table 3 | Main parts of a generated function module. ....                                      | 66 |
| Table 4 | Expected-failure test categories. ....                                               | 85 |
| Table 5 | Benchmark environment ....                                                           | 87 |
| Table 6 | Benchmark programs used in the performance evaluation. ....                          | 88 |

|                                                           |    |
|-----------------------------------------------------------|----|
| Table 7 Overview of possible future-work directions. .... | 94 |
|-----------------------------------------------------------|----|

## List of images

|                                                                                                                                                                                                                                                                                                 |    |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----|
| Figure 1 Example module dependency graph. Arrows point from a module to a module it depends on. Loading <code>main</code> requires both <code>math</code> and <code>core</code> , but <code>core</code> is initialized only once even though it is reachable through two dependency paths. .... | 50 |
| Figure 2 Overview of the TinyClojure compiler pipeline. Boxes represent compiler phases, while labels between them represent intermediate program representations. The generated LLVM IR modules are either added to the running JIT session or written as AOT artifacts. ....                  | 59 |
| Figure 3 TinyClojure slowdown relative to JVM Clojure using the default optimized TinyClojure configuration. Bars show slowdown factors computed from average runtimes; error bars show standard deviation converted to slowdown units. ....                                                    | 89 |
| Figure 4 Effect of integer-cache range on benchmark runtime. Bars show average runtime; error bars show standard deviation. ....                                                                                                                                                                | 90 |
| Figure 5 Effect of LLVM optimization level on benchmark runtime. Bars show average runtime; error bars show standard deviation. ....                                                                                                                                                            | 91 |
| Figure 6 Effect of direct linking on benchmark runtime. Bars show average runtime; error bars show standard deviation. ....                                                                                                                                                                     | 92 |

## List of abbreviations

- AOT*** – ahead-of-time 51, 51, 51, 51, 51, 52, 52, 64, 64, 69, 86, 86, 86, 86, 87, 97, 99, 107
- API*** – application programming interface 16, 16, 36, 38, 68
- AST*** – abstract syntax tree 5, 5, 5
- IR*** – intermediate representation 1, 2, 8, 12, 12, 12, 12, 13, 13, 13, 13, 13, 14, 14, 14, 15, 15, 16, 38, 38, 38, 39, 41, 41, 42, 45, 48, 60, 60, 62, 63, 65, 65, 66, 68, 68, 68, 69, 70, 71, 74, 74, 80, 80, 84, 86, 92, 93, 94, 99, 99, 100, 100
- JIT*** – just-in-time 12, 51, 51, 64
- JVM*** – Java Virtual Machine 1, 1, 1, 1, 2, 3, 3, 6, 6, 6, 6, 6, 6, 6, 7, 7, 7, 8, 8, 8, 8, 8, 8, 8, 9, 9, 10, 11, 11, 11, 12, 12, 12, 13, 18, 18, 47, 66, 83, 83, 83, 83, 84, 84, 87, 87, 87, 88, 88, 88, 88, 88, 88, 89, 89, 89, 89, 93, 99, 99, 99, 100, 107
- REPL*** – read-eval-print loop 9, 35, 35, 36, 51, 51, 55, 69, 69, 69, 82, 84, 84, 85
- SSA*** – static single assignment 14, 14, 14, 15
- WSL*** – Windows Subsystem for Linux 87

# Introduction

Clojure is a modern dialect of Lisp designed as a hosted language. Its primary implementation runs on the Java Virtual Machine (JVM), which provides a mature runtime environment, automatic memory management, class loading, exception handling, and access to the Java ecosystem. These services make the JVM a practical host platform, but they also shape the implementation model of the language.

LLVM offers a different kind of target. It provides a low-level intermediate representation, optimization infrastructure, and native code generation for multiple platforms. Unlike the JVM, however, LLVM is not a managed language runtime. It does not provide built-in support for high-level language features such as dynamic values, Vars, function objects, closures, lists, or garbage collection. A language implementation targeting LLVM must therefore represent these features explicitly.

Implementing a complete Clojure compiler targeting LLVM would be far beyond the scope of a bachelor thesis. Clojure includes a large standard library, persistent data structures, advanced concurrency primitives, Java interoperability, a rich macro system, and many details tied to its hosted runtime model. This thesis therefore focuses on a restricted language called TinyClojure. TinyClojure is a small dynamically typed Lisp-like language inspired by Clojure. It preserves selected core features, including symbolic forms, lists, Vars, first-class functions, closures, conditionals, recursion, and a simplified macro system.

The goal of this thesis is to design and implement a compiler for TinyClojure that generates LLVM intermediate representation (IR), together with a supporting runtime system implemented in C++. The runtime provides representations of dynamic values, function objects, closures, global Vars, module loading, and memory management. The implementation supports both just-in-time execution and ahead-of-time compilation of modules that can be loaded at runtime.

The resulting system is evaluated in terms of correctness, expressiveness, and performance. The evaluation focuses on whether the implemented subset behaves as expected, which language features it can express, and what trade-offs arise from targeting LLVM directly instead of relying on the JVM runtime.

The rest of this thesis is structured as follows. Chapter 1 introduces the background needed to understand the implementation, including compiler phases, Lisp and Clojure concepts, the JVM, Jank, and LLVM. Chapter 2 defines the TinyClojure language subset. Chapter 3 describes the runtime system. Chapter 4 presents the compiler implementation and LLVM IR generation. Chapter 5 evaluates the resulting system, summarizes the results and discusses possible future work.

# Chapter 1

## Background

### 1.1 Lisp and Clojure

Lisp [1] is a family of programming languages based on symbolic expressions. Lisp programs are written as *forms*. A form is a piece of program text that can be evaluated by the language implementation. It may be an atomic value, such as a number, string, or symbol, or a compound form represented as a list.

A compound form is usually written as a parenthesized list. For example, the expression `(+ 1 2)` is read as a list containing the symbol `+` and the numbers `1` and `2`. This close correspondence between program text and ordinary language data is commonly referred to as homoiconicity [2]. In this thesis, this property is important because the TinyClojure reader produces data structures that are later analyzed by the compiler.

Clojure is a modern Lisp dialect. It keeps the uniform expression-based syntax and macro system of Lisp, while adding its own runtime model, immutable persistent data structures [3], namespace system, hosted execution model, and interoperability with its host platform. Its primary implementation targets the JVM [4]. This hosted execution model is one reason why the JVM is discussed separately in Section 1.4.

For example, the following Clojure code defines a function and calls it:

```
(defn square [x]
  (* x x))

(square 5)
```

Both the function definition and the function call are written as forms. The first element of each list determines how the rest of the form is interpreted: `defn` introduces a function definition, while `square` is used as a function call.

### 1.2 Source language model

TinyClojure is the restricted Clojure-inspired language implemented in this thesis. At a high level, it follows the Lisp model described in the previous

section: programs are written as forms, compound forms are represented as lists, and most constructs evaluate to values.

The language is expression-oriented and dynamically typed. This means that constructs such as conditionals, local bindings, and function calls produce values, while many checks that would be static in other languages are performed at runtime.

Functions are first-class values. They can be stored, passed as arguments, returned from other functions, and captured by inner functions. This makes closures one of the central language features considered by the implementation.

A precise definition of the implemented language subset is given in Chapter 2.

## 1.3 Standard architecture of modern compilers

Compilers are commonly described as a sequence of phases that transform a source program into a target representation [5]. The exact organization differs between implementations, but the usual phases include lexical analysis, syntactic analysis, semantic analysis, optimization, and code generation.

Each phase works on a representation produced by the previous one. Lexical analysis divides source text into tokens, syntactic analysis determines the structure of the program, semantic analysis attaches language-specific meaning to this structure, optimization improves an intermediate representation without changing the intended behavior, and code generation produces the selected target representation.

The exact boundaries between these phases are not always strict. Some languages combine several of them, while others insert additional phases that are specific to the source language.

### 1.3.1 Lexical analysis and reading

Lexical analysis is the process of reading source text and dividing it into basic units, usually called tokens. In languages with more conventional syntax, tokens include identifiers, keywords, literals, operators, delimiters, and punctuation. For example, the expression `x + 1` may be split into an identifier `x`, an operator `+`, and an integer literal `1`.

The output of lexical analysis is usually consumed by a parser. Tokens often also carry additional information, such as their textual value and source location, which is useful for error reporting.

In Lisp-family languages, including Clojure, the first phase is often described as reading [2] rather than ordinary lexical analysis. The reader

does not only produce a flat sequence of tokens; it constructs data structures representing the source forms, such as symbols, literals, and lists. For example, the form `(+ 1 2)` can be read as a list containing the symbol `+` and the integer literals `1` and `2`. This is important for TinyClojure, because macroexpansion and semantic analysis operate on these read forms.

### 1.3.2 Syntactic analysis

Syntactic analysis determines the structure of a program. In many languages, this means parsing a token stream according to a grammar and producing an abstract syntax tree (AST). The AST represents the hierarchical structure of the program while omitting details that are not relevant for later phases.

For example, a parser for a C-like language may represent a function definition as a node containing the function name, parameters, return type, and body. Expressions and statements inside the body are represented by nested nodes.

In Clojure-like languages, much of the nesting is already determined by the reader. For example, parenthesized forms are read as lists. However, these forms are not yet semantically meaningful. A list may represent a function call, a special form, a macro invocation, or data quoted by the program. Therefore, TinyClojure still needs a later analysis phase that decides how each form should be interpreted or compiled.

### 1.3.3 Semantic analysis

Semantic analysis checks and enriches the syntactic representation of the program. It resolves names, verifies language-specific rules, and produces a representation suitable for code generation.

In statically typed languages, semantic analysis often performs type checking before the program is executed. In dynamically typed languages, many type checks are delayed until runtime. Nevertheless, semantic analysis is still necessary. For example, the compiler must distinguish local variables from global variables, detect invalid special forms, resolve function arguments, and determine which variables are captured by closures.

In TinyClojure, semantic analysis transforms read and macroexpanded forms into an internal AST used by later compiler phases. It resolves symbols, distinguishes special forms from ordinary function calls, and rejects forms that are not valid in the implemented language subset. The language subset itself is defined in Chapter 2, while the implementation of semantic analysis is described in Section 4.3.

### 1.3.4 Optimization

An optimization is a transformation that preserves the observable behavior of a program while improving some property of the generated code, most commonly execution time, memory usage, or code size. Optimizations are often performed on an intermediate representation rather than directly on source code.

Common examples [6] include constant folding, dead code elimination, inlining, and tail-call optimization. Constant folding replaces expressions whose values are known at compile time with their result. Dead code elimination removes code that cannot affect the result of the program. Inlining replaces a function call with the body of the called function. Tail-call optimization can replace certain recursive calls with jumps, avoiding additional stack usage.

In this thesis, optimizations are discussed again when describing the TinyClojure compiler and its LLVM backend.

### 1.3.5 Code generation

Code generation translates the analyzed representation of the program into the selected target representation. Depending on the compiler, the target may be assembly code, machine code, bytecode, or another intermediate representation.

The choice of target determines which services the compiler can rely on and which services must be implemented by the language runtime. For example, a managed virtual machine may provide automatic memory management and object support, while a lower-level target may require these features to be represented explicitly.

TinyClojure uses LLVM as its final compiler backend; LLVM itself is introduced in Section 1.7.

## 1.4 Java and the JVM

The JVM is a virtual execution environment originally designed for the Java programming language [4]. Java source files are compiled into `.class` files containing JVM bytecode. The JVM loads these class files, verifies them, links them with required classes, and executes their bytecode.

The JVM provides a portable abstraction over the underlying machine. A program compiled to JVM bytecode can run on different operating systems and processor architectures as long as a compatible JVM implementation is available. This portability is one of the reasons why the JVM is used as a host platform for several languages besides Java, including Clojure, Scala, and

Kotlin. These languages do not only target JVM bytecode; they also commonly reuse parts of the Java runtime environment, such as class loading, garbage collection, threads, exceptions, reflection, and the Java standard library.

For Clojure, the JVM is therefore not merely a code generation target. It also provides important runtime services and access to the Java ecosystem. When targeting LLVM directly, these services are no longer provided by the target platform. The implementation cannot assume a built-in object model, garbage collector, class loader, dynamic dispatch mechanism, or standard library corresponding to Clojure's runtime needs. Features such as dynamic values, function objects, closures, global Vars, module loading, and memory management must therefore be represented explicitly by the compiler and runtime.

### 1.4.1 Execution model

The JVM is a stack-based virtual machine. Most bytecode instructions operate on an operand stack. Instructions push values onto the stack, consume values from the stack, or move values between the operand stack and local variables.

For example, consider the following Java code:

```
int x = 2;
int y = 4;
int z = 6;
System.out.println(x + y * z);
```

A simplified bytecode sequence for the arithmetic expression may look as follows:

```
iconst_2
istore_1
iconst_4
istore_2
bipush 6
istore_3
getstatic #7    // Field java/lang/System.out:Ljava/io/PrintStream;
iload_1
iload_2
iload_3
imul
iadd
invokevirtual #13 // Method java/io/PrintStream.println:(I)V
```

The instructions `iconst_2`, `iconst_4`, and `bipush 6` push integer constants onto the operand stack. The corresponding `istore` instructions store these values into local variable slots. Later, the `iload` instructions load the local variables back onto the operand stack. The `imul` instruction multiplies the

two topmost integer values and pushes the result, and `iadd` adds the next two integer values.

The instruction `getstatic #7` loads the static field `java.lang.System.out`, which is a `PrintStream` object. The number `#7` is an index into the constant pool of the current class. Similarly, `invokevirtual #13` calls the instance method `println` described by another constant-pool entry. The exact numeric indices are not meaningful by themselves; they depend on the contents and ordering of the constant pool generated for the class.

This stack-based bytecode model differs from LLVM IR, introduced in Section 1.7, where intermediate values and control flow are represented explicitly. This distinction is relevant because TinyClojure targets LLVM rather than JVM bytecode.

### 1.4.2 Class loading and initialization

Code executed by the JVM is organized into classes. For Java programs, source files are compiled into class files, and each class file defines a class or interface together with its bytecode, constant pool, fields, and methods. Languages targeting the JVM, including Clojure, also ultimately produce classes that are loaded and initialized by the JVM.

A class is loaded by a class loader when it is first needed. Loading is followed by linking, which includes verification and preparation of the class, and then by initialization. During initialization, the JVM executes the class initializer method, commonly referred to as `<clinit>`, if the class defines one.

In Java source code, class initialization corresponds to the initialization of static fields and execution of static initializer blocks:

```
static int x = computeValue();

static {
    initializeSomething();
}
```

These actions are compiled into bytecode that is executed when the class is initialized.

## 1.5 JVM Clojure

Clojure is a hosted Lisp dialect whose primary implementation targets the JVM [7]. As a hosted language, Clojure reuses many services provided by its host platform, including class loading, garbage collection, exceptions, threads, and access to Java libraries. At the same time, Clojure defines its own lan-

guage-level runtime model, including namespaces, Vars, first-class functions, immutable collections, and macros.

Clojure programs are compiled to JVM bytecode, but the language remains dynamic. Vars may be redefined, code can be evaluated at runtime, and many operations are resolved through runtime objects. This section describes the parts of JVM Clojure that are most relevant for TinyClojure: namespaces and Vars, function invocation, closures, and the compilation model.

### 1.5.1 Namespaces and Vars

Clojure code is organized into namespaces. A namespace maps symbols to Vars and provides a way to refer to definitions from other namespaces. A Var is a mutable reference object that usually stores a global value, such as a function or constant. [8]

Although Vars are mutable, top-level Vars are commonly used as stable definitions: they are created and bound during namespace loading and then mostly accessed by dereferencing. This convention allows Clojure programs to behave similarly to programs with immutable top-level definitions in ordinary use. However, the mutability remains observable. A Var can be redefined, for example during interactive development at the read-eval-print loop (REPL), and code that performs an ordinary Var dereference will see the updated root value.

During compilation, a symbol can resolve either to a local binding or to a Var. For example, in `(let [x 1] (+ x 2))`, the symbol `x` refers to a local binding, while `+` usually resolves to Var `clojure.core/+`. This distinction is important because local variables can be accessed directly, while Vars introduce a level of indirection: the Var must be dereferenced to obtain its current root value.

The indirection through Vars is one of the mechanisms that makes Clojure dynamic. A Var can be rebound or redefined, and code that invokes a Var through ordinary dynamic dispatch observes the current value of that Var. Direct linking can remove this indirection in selected cases, but doing so changes the behavior with respect to later redefinitions.

For example:

```
(def a 3)

(defn print-a []
  (println a))

(print-a)

(def a 4)

(print-a)
```

The symbol `a` inside `print-a` refers to a Var, not to a fixed local value captured when the function is defined. The function therefore dereferences the Var when it is called. The first call to `print-a` prints 3, while the second call prints 4, because the root value of the Var has changed.

### 1.5.2 Function invocation

Clojure represents callable values using the `IFn` Java interface. Function objects implement this interface, but they are not the only callable objects. Vars, keywords, maps, sets, and other objects can also be invoked according to their runtime behavior.

For ordinary user-defined functions, the compiler generates a JVM class representing the function. The generated class implements the supported arities by defining corresponding `invoke` methods. Calls with unsupported arities result in an arity exception. Conceptually, a function such as

```
(defn foo
  ([] 0)
  ([x] x))
```

is represented by a generated class with methods for the zero-argument and one-argument cases.

In addition to instance `invoke` methods, Clojure may also generate static methods such as `invokeStatic`. These methods allow selected calls to avoid ordinary interface dispatch. When *direct linking* is enabled and the compiler can determine the target function, a call can be compiled directly to the generated static method instead of dereferencing a Var and invoking the resulting function object.

This optimization is Clojure-specific terminology, but it is related to a broader family of optimizations for reducing dynamic dispatch overhead. In general compiler terminology, replacing a dynamically selected call target with a known direct call is related to early binding, static binding, and devirtualization. Devirtualization replaces an indirect or virtual call with a direct call when the concrete target is known [9]. Dynamic-language implementations

may also use call-site specialization, where generated code is specialized using information about a particular call site [10].

Another related technique is inline caching, originally used in Smalltalk implementations [11]. Inline caching is different from direct linking: it keeps the call site dynamic, but caches previously observed lookup results at runtime. Direct linking, as used by Clojure and TinyClojure, fixes the selected target during compilation.

### 1.5.3 Closures

Functions may capture local variables from their surrounding lexical scope. Such functions are called closures. For example:

```
(defn make-adder [x]
  (fn [y]
    (+ x y)))
```

The inner function uses the variable `x`, even though `x` belongs to the outer function. JVM Clojure represents such a closure by generating a class with fields for the captured values. When the closure object is constructed, the captured values are passed to its constructor and stored in these fields. Later, the closure's `invoke` method reads the fields when evaluating the body.

TinyClojure uses a different low-level representation, but it must solve the same problem: if a function escapes the scope in which it was created, the values it uses from that scope must remain available.

### 1.5.4 Compilation model

Clojure code can be compiled ahead of time into JVM class files. A compiled namespace is represented by generated classes. One class acts as a namespace loader: it initializes constants and Var references and executes top-level forms in the namespace. Additional generated classes represent functions and closures.

This model relies on the JVM class-loading and initialization mechanism described in Section 1.4.2. Loading a generated namespace class can initialize static fields containing references to runtime objects such as symbols, constants, Vars, and function objects. It can also execute code that loads the namespace.

A simplified namespace loader therefore performs two kinds of work. First, it initializes static fields containing references to runtime objects. Second, it executes the top-level forms of the namespace, for example by binding function objects to Vars or evaluating top-level expressions.

This model is relevant for TinyClojure because TinyClojure also separates compiled function bodies from module-level initialization. Function bodies are compiled into separate LLVM modules, while file-level modules load dependencies, declare Vars, initialize the root values stored in those Vars, and execute top-level forms.

## 1.6 Jank

Jank [12] is a native Clojure dialect implemented in C++ and hosted on LLVM. Its goal is to preserve the interactive and value-oriented style of Clojure while using a native runtime and providing direct interoperability with C++ code. Unlike JVM Clojure, whose host platform is the JVM, Jank uses LLVM and native code generation as its host environment.

Jank explores a similar design space as the goal of this thesis: implementing a Clojure-like language outside the JVM while retaining dynamic language features such as first-class functions, runtime values, macros, and interactive evaluation. However, the goals of Jank and TinyClojure are different. Jank aims to be a practical Clojure dialect with strong compatibility and native interoperability, while TinyClojure is a small educational subset designed to study the implementation of a dynamically typed Lisp-like language on top of LLVM.

A major feature of Jank is C++ interoperability. Since Jank is designed to call and embed native C++ code, its compiler must handle not only Clojure-like forms, but also interactions with C++ declarations, types, overloads, name lookup, and exceptions. This makes the implementation substantially more complex than a compiler for a closed language subset.

This also influences Jank's compilation model. Earlier versions of Jank experimented with direct generation of LLVM IR. However, this approach makes C++ interoperability difficult, because it would require access to the C++ compiler's semantic knowledge.

For this reason, Jank moved toward a pipeline where Jank code is lowered to C++ code rather than directly to LLVM IR. The generated C++, together with any embedded or referenced C++ code, is then processed by Clang. Clang performs the C++ semantic analysis and lowers the result to LLVM IR, after which LLVM provides optimization, native code generation and just-in-time (JIT) execution.

TinyClojure deliberately takes a smaller approach. It does not implement C++ interoperability and does not aim for full Clojure compatibility. Instead, it compiles a restricted set of Clojure-like forms directly to LLVM IR and provides only the runtime support needed for this subset. This makes it possible to study the core issues – dynamic values, Vars, function objects,

closures, macroexpansion, and module loading – without implementing the full host interop model required by Jank.

## 1.7 LLVM

LLVM [13] is a modular compiler infrastructure centered around LLVM IR, a low-level intermediate representation designed for program analysis, optimization and machine-code generation. A compiler front end can translate a source language into LLVM IR, after which LLVM can apply optimization passes and generate code for different target architectures. LLVM optimization passes operate on LLVM IR and can perform transformations such as dead-code elimination, inlining, constant propagation, and simplification of control flow before target-specific code generation.

Unlike the JVM, LLVM is not a managed language runtime. It does not define a built-in object model or dynamic dispatch mechanism, nor does it provide a ready-to-use garbage collector or a language standard library. These services must be provided by the language implementation when they are needed.

### 1.7.1 LLVM IR

LLVM IR is the intermediate representation used by LLVM. It is lower-level than source languages such as Clojure, but still independent of a specific processor architecture. It represents programs using modules, functions, basic blocks, instructions, typed values, and explicit control flow.

For example, consider the following C program:

```
#include <stdio.h>

int main(int argc, char **argv) {
    printf("Hello, argc=%d\n", argc);
    return 0;
}
```

■ **Listing 1** C program used as a running LLVM example.

The C program in Listing 1 can be represented by the simplified LLVM IR in Listing 2.

```
@fmt = private unnamed_addr constant [17 x i8] c"Hello,  
argc=%d\0A\00"  
  
declare i32 @printf(ptr, ...)  
  
define i32 @main(i32 %argc, ptr %argv) {  
entry:  
    %fmt_ptr = getelementptr inbounds [17 x i8], ptr @fmt, i32 0, i32  
0  
    call i32 (ptr, ...) @printf(ptr %fmt_ptr, i32 %argc)  
    ret i32 0  
}
```

■ **Listing 2** Simplified LLVM IR module for a C program that calls `printf`.

This example shows several basic LLVM IR constructs. Names beginning with `@` refer to global values, such as global constants or functions. Names beginning with `%` refer to local static single assignment (SSA) values inside a function. The function `main` contains one basic block named `entry`. The `getelementptr` instruction computes a pointer to the first character of the global string constant, the `call` instruction invokes the external function `printf`, and the `ret` instruction returns from `main`.

LLVM IR is typed. Values have LLVM types such as `i64`, `i1`, or `ptr`. However, these LLVM types are not the same as source-language types. In TinyClojure, most user-visible values are represented uniformly as pointers to runtime objects. For example, an integer value in the source language is usually not kept as a raw LLVM `i64` throughout the program; generated code obtains a pointer to a runtime object representing that integer.

Operations that depend on TinyClojure semantics are therefore emitted as calls to runtime functions. This includes allocation of boxed values, arithmetic over dynamic values, access to global variables, function invocation, and list operations. The concrete runtime representation of values is described in Section 3.1.

### 1.7.2 SSA and phi nodes

LLVM IR uses SSA form [14]. In SSA form, each named value is assigned exactly once. When control-flow paths merge, LLVM uses phi nodes to select the value corresponding to the predecessor block from which control arrived.

For example, consider the following C-like conditional:

```
int x;  
if (condition)  
    x = 1;  
else  
    x = 2;
```

After lowering to SSA form, this can be represented using separate basic blocks for the two branches and a merge block:

```
entry:  
    br i1 %condition, label %then, label %else  
  
then:  
    br label %merge  
  
else:  
    br label %merge  
  
merge:  
    %x = phi i32 [1, %then], [2, %else]
```

■ **Listing 3** Example usage of phi nodes in LLVM IR

The phi node defines `%x` as 1 if control arrived from the `then` block and as 2 if control arrived from the `else` block. In this simplified example, the incoming values are integer constants. More generally, a phi node selects between values produced in predecessor blocks; these values may be constants, instruction results, loaded values, or pointers returned by function calls.

The same control-flow structure is useful for expression-oriented languages. A TinyClojure expression such as `(if condition 1 2)` also produces a value, so its branches can be lowered to separate basic blocks whose results are merged with a phi node. The actual TinyClojure compiler usually works with pointers to runtime objects rather than raw `i32` values, but the control-flow principle is the same.

### 1.7.3 LLVM modules

The LLVM IR shown in Listing 2 is also an example of an LLVM module. A module roughly corresponds to an individual translation unit: it contains the IR produced for one separately compiled part of a program. A module may contain function definitions, global variables, constants, type definitions, and declarations of external symbols.

In Listing 2, `@fmt` is a global constant defined in the module, `@printf` is an external function declaration, and `@main` is a function definition. This distinction between definitions and declarations is important for TinyClojure,

because generated modules often call runtime helper functions or functions defined in other modules.

TinyClojure uses this structure by compiling source files and generated functions into separate LLVM modules. File-level modules contain top-level initialization code, while function modules contain compiled function bodies. Runtime module loading is described in Section 2.7, and function modules are described in Section 4.4.1.

#### 1.7.4 LLVM C++ API

TinyClojure constructs LLVM IR using the LLVM C++ application programming interface (API). Instead of printing LLVM IR text directly, the compiler creates modules, functions, basic blocks, and instructions through C++ objects.

The module structure shown in Listing 2 can also be constructed programmatically using the LLVM C++ API:

```

llvm::LLVMContext context;
auto module = std::make_unique<llvm::Module>("example", context);
llvm::IRBuilder<> builder(context);

auto *i32 = llvm::Type::getInt32Ty(context);
auto *ptr = llvm::PointerType::getUnqual(context);

// Create a private global string constant.
auto *fmt = builder.CreateGlobalString(
    "Hello, argc=%d\n",
    "fmt",
    0,
    module.get());

// Declare: i32 @printf(ptr, ...)
auto *printf_type = llvm::FunctionType::get(i32, {ptr}, true);
auto *printf_fn = llvm::Function::Create(
    printf_type,
    llvm::Function::ExternalLinkage,
    "printf",
    module.get());

// Define: i32 @main(i32 %argc, ptr %argv)
auto *main_type = llvm::FunctionType::get(i32, {i32, ptr}, false);
auto *main_fn = llvm::Function::Create(
    main_type,
    llvm::Function::ExternalLinkage,
    "main",
    module.get());

auto arg_it = main_fn->arg_begin();
llvm::Value *argc = &*arg_it++;
argc->setName("argc");
llvm::Value *argv = &*arg_it;
argv->setName("argv");

auto *entry = llvm::BasicBlock::Create(context, "entry", main_fn);
builder.SetInsertPoint(entry);

auto *zero = llvm::ConstantInt::get(i32, 0);
auto *fmt_ptr = builder.CreateInBoundsGEP(
    fmt->getValueType(),
    fmt,
    {zero, zero},
    "fmt_ptr");

builder.CreateCall(printf_fn, {fmt_ptr, argc});
builder.CreateRet(llvm::ConstantInt::get(i32, 0));

```

■ **Listing 4** Constructing the module from Listing 2 using the LLVM C++ API.

Listing 4 creates the same module structure programmatically. `LLVMContext` owns global LLVM state, `Module` owns the global string, the `printf` declaration, and the `main` function, `FunctionType` describes function signatures, `Function::Create` inserts functions into the module, `BasicBlock::Create` creates the entry block, and `IRBuilder` emits instructions.

## 1.8 Other native implementations of dynamic languages

Native and ahead-of-time compilation of dynamic languages has been explored by several existing systems. Bigloo [15] is an implementation of Scheme with an optimizing compiler that can generate C code or JVM bytecode. It is an example of a Lisp-family language implementation that uses ahead-of-time compilation and existing lower-level targets rather than interpreting source programs directly.

Similar approaches are used for Python. Cython [16] is an optimizing compiler for Python and the Cython language. It generates C code by default, and can also generate C++ code when compiling modules that use C++ features or request C++ output.

These systems differ in language scope and compatibility goals, but they share a common implementation pattern: programs written in a dynamic language are lowered to an existing target such as C, C++, or JVM bytecode. These targets are attractive because they allow an implementation to reuse mature compilers, optimizers, linkers, runtime libraries, debugging tools, and platform support.

# The TinyClojure specification

This chapter defines TinyClojure, the language implemented in this thesis. TinyClojure is a restricted dynamically typed Lisp-like language inspired by Clojure. It is not intended to be a complete or source-compatible implementation of Clojure. Instead, the subset is chosen to preserve the core implementation challenges relevant to this work: reading symbolic forms, resolving variables, compiling first-class functions and closures, supporting recursion, performing macroexpansion, and representing dynamic values at runtime.

TinyClojure programs are composed of forms. A form is a syntactic unit that can be evaluated by the language implementation. A form may be atomic, such as an integer, string, symbol, or boolean, or compound, such as a list. During execution, forms evaluate to runtime values. Some forms evaluate to themselves, while others are interpreted as variable references, special forms, macro invocations, or function applications.

The implemented subset supports integers, floating-point numbers, characters, strings, symbols, booleans, `nil`, lists, global Var objects, first-class functions, closures, conditionals, local bindings, recursion, and a simplified macro system. Features of full Clojure such as persistent vectors, maps and sets, protocols, multimethods, Java interoperability, concurrency primitives, metadata, lazy sequences, and full namespace management are outside the scope of this thesis.

## 2.1 Reader syntax

TinyClojure source code is first processed by a reader. The reader transforms source text into TinyClojure data structures such as literals, symbols, and lists. Unlike languages with complex infix syntax and operator precedence, TinyClojure uses a uniform Lisp-style syntax where compound forms are written as lists.

The reader only recognizes the surface structure of the program. For example, it recognizes `x` as a symbol, but it does not decide what value the symbol refers to. Similarly, it recognizes a parenthesized form as a list, but

it does not decide how that list should be interpreted. These meanings are assigned by later phases of the language implementation.

### 2.1.1 Whitespace and comments

Whitespace separates tokens and is otherwise ignored. The comma `,` is also treated as a whitespace character. Line comments begin with `;` and continue until the end of the line. Comments are ignored by the reader and do not produce runtime values.

### 2.1.2 Integer literals

Integer literals represent signed 64-bit integers. They are written in decimal notation and may optionally start with `+` or `-`:

```
0
123
+123
-123
```

If an integer literal does not fit into the supported 64-bit signed range, reading the source program fails with a read error.

### 2.1.3 Floating point literals

Floating-point literals represent 64-bit IEEE 754 floating-point values [17]:

```
1.0
-2.5
```

Scientific notation, such as `3.1e4`, is not supported.

### 2.1.4 Character literals

Character literals represent individual characters. All character literals begin with a backslash. A literal consisting of a backslash followed by a single character represents that character directly:

```
\a
\0
\)
\'
\"
```

The backslash character itself is written as `\\`.

Some control and whitespace characters are written using named character literals [18]:

```
\newline ; linefeed, 0x0A
\space  ; space, 0x20
\tab    ; horizontal tab, 0x09
\return ; carriage return, 0x0D
\backspace ; backspace, 0x08
```

TinyClojure supports hexadecimal character literals. These are written as `\x` followed by one or more hexadecimal digits. The value must fit into one byte, so the accepted range is from `\x0` to `\xFF`.

```
\x0 ; the NUL character, 0x00
\x41 ; the character A, 0x41
\xff ; the byte value 0xFF
```

TinyClojure supports octal character literals. These are written as `\o` followed by one or more octal digits. The value must fit into one byte, so the accepted range is from `\o0` to `\o377`.

```
\o0 ; the NUL character, 0x00
\o101 ; the character A, 0x41
\o377 ; the byte value 0xFF
```

Any other multi-character character literal is rejected by the reader. For example, `\foo`, `\o8`, `\o400`, `\x100`, and `\u123` are invalid.

### 2.1.5 String literals

String literals are enclosed in double quotes: `"Hello, world!"`. Inside strings, special characters double quote and backslash must be escaped. Additionally, a subset of other characters can be escaped, similarly to other languages:

```
"\n" ; the linefeed character 0x0A
"\t" ; the tab character 0x09
"\r" ; the carriage return character 0x0D
"\b" ; the backspace character 0x08
```

Other escape sequences are rejected by the reader.

### 2.1.6 Symbolic value literals

The literal `nil` represents the absence of a value. It is distinct from `false`, numeric zero, the `0x00` ASCII character, the empty string, the empty list or similar. Some sequence operations treat `nil` in the same way as an empty list. This is a property of those operations, not an equality between `nil` and the empty list.

The literals `true` and `false` represent boolean values.

### 2.1.7 Symbols

Symbols are names used to refer to local bindings, Vars, special forms, macros or functions. Examples of symbols are:

```
x  
main  
+  
my-function  
symbol?
```

Symbols may contain letters, digits, and selected punctuation characters such as +, -, \*, /, ?, !, <, =, >, \_, %, &, \$, #, and :.

A symbol may not start with a character sequence that is recognized as another literal, such as a number, string, keyword, or character literal.

### 2.1.8 Lists

Lists begin with '(' and end with ')'. They can contain arbitrary values, optionally separated with whitespaces, including other lists:

```
() ; empty list  
(+ 1 2)  
(if true 1 2)
```

### 2.1.9 Quote

The quote syntax prevents evaluation of a form and returns the form as data:

```
'abc  
'(1 2 3)
```

These examples are equivalent to:

```
(quote abc)  
(quote (1 2 3))
```

### 2.1.10 Syntax quote

Syntax quote is written using ```. It is similar to quote, but it allows selected forms inside the quoted form to be evaluated when the syntax-quote expression is evaluated. Such nested forms are called subforms.

For example, in `(if condition (+ x 1) 0)`, the forms `if`, `condition`, `(+ x 1)`, and `0` are subforms of the outer list.

TinyClojure supports two reader forms inside syntax quote: `unquote`, written `~form`, and `unquote-splicing`, written `~@form`.

Inside a syntax-quoted form, ordinary subforms are quoted by default. In `~form`, the form after `~` is evaluated and the resulting value is inserted into the generated form. In `~@form`, the form after `~@` is evaluated, but its result must be *spliceable*. A spliceable value is a value whose elements can be inserted into another list; in TinyClojure, only lists and `nil` are spliceable. If the result is a list, its elements are inserted individually into the surrounding list. If the result is `nil`, no elements are inserted.

As an example, consider a form where `a` is bound to 42:

```
(let* (a 42)
  `(+ ~a 1))
```

This evaluates to a list equivalent to:

```
(+ 42 1)
```

Unquote-splicing inserts the elements of a list into the surrounding list. For example:

```
(let* (args (list 1 2 3))
  `(+ ~@args))
```

This evaluates to a list equivalent to:

```
(+ 1 2 3)
```

By contrast, ordinary unquote inserts the list itself as one element:

```
(let* (args (list 1 2 3))
  `(+ ~args))
```

This evaluates to a list equivalent to:

```
(+ (1 2 3))
```

TinyClojure also supports auto-gensym symbols. A symbol ending with `#` inside a syntax-quoted form is replaced by a generated unique symbol. Repeated uses of the same auto-gensym symbol inside one syntax-quoted form are replaced by the same generated symbol.

For example:

```
`(let* (tmp# 1)
  tmp#)
```

expands to a form equivalent to:

```
(let* (tmp__123 1)
  tmp__123)
```

Auto-gensym is useful when writing macros that introduce local bindings. It helps avoid accidental conflicts with names used by the code into which the macro is expanded. For example, consider a macro that introduces an internal temporary binding:

```
(defmacro with-internal-temp (& body)
  `(let* (tmp# "internal")
    ~@body))
```

If this macro is used in a scope that already contains a binding named `tmp`,

```
(let* (tmp "user")
  (with-internal-temp
    (println tmp)))
```

the macro expands to a form equivalent to:

```
(let* (tmp "user")
  (let* (tmp__123 "internal")
    (println tmp)))
```

The generated temporary name does not shadow the user's binding `tmp`. Without auto-gensym, a macro that introduced a plain binding named `tmp` could accidentally change the meaning of the user's code.

## 2.2 Runtime values

TinyClojure is dynamically typed. Runtime type information is associated with values, not with names. The same local binding, function parameter or global reference may therefore contain values of different types at different times or during different executions of the program.

The runtime values supported by TinyClojure are:

- `nil`
- booleans
- signed 64-bit integers
- 64-bit floating-point numbers
- characters
- strings
- symbols
- lists
- global reference objects, called Vars
- functions and closures

Most literal values evaluate to themselves. Lists are the main exception: an unquoted list is interpreted as a form to be evaluated, while a quoted list evaluates to list data.

### 2.2.1 Nil and booleans

The value `nil` represents the absence of a value. It is distinct from `false`, numeric zero, the empty string, and the empty list. In conditional contexts, only `nil` and `false` are falsy; all other values are truthy.

### 2.2.2 Numbers

TinyClojure supports signed 64-bit integers and 64-bit floating-point numbers. Arithmetic operations are dynamically checked with respect to operand types. If an operation receives values of unsupported types, a runtime error is signalled.

Signed integer overflow is not part of the defined TinyClojure semantics. Programs that rely on overflow behavior are outside the supported subset.

### 2.2.3 Lists

Lists are sequential collections. They are used both as ordinary data and as the representation of compound source forms. A quoted list evaluates to list data. An unquoted non-empty list is interpreted as a form and evaluated to the rules described in Section 2.3.

### 2.2.4 Vars

A Var is a mutable global reference object. It stores a value that can be read or updated by the program. Top-level definitions create or update Vars.

Evaluating a symbol that resolves to a Var reads the value stored in the Var. The `var` special form, described in Section 2.4.8, can be used to obtain the Var object itself instead of reading its stored value.

For example:

```
(def x 42)

x          ; evaluates to 42
(var x)    ; evaluates to the Var object for x
```

Here, `x` reads the value stored in the Var, while `(var x)` returns the Var object itself.

### 2.2.5 Functions and closures

Functions are first-class values. They can be stored in Vars, passed as arguments, returned from other functions, and invoked.

A closure is a function that captures values from its surrounding lexical scope. Captured values remain available even after the surrounding function has returned.

## 2.3 Evaluation model

Evaluation maps forms to runtime values. Every form either evaluates to a value or signals an error.

### 2.3.1 Self-evaluating forms

The following forms evaluate to themselves: `nil`, booleans, numbers, characters and strings.

### 2.3.2 Symbol resolution

A symbol is evaluated by resolving it in the current environment. Local bindings have priority over global Vars. If a symbol resolves to a local binding, evaluating it returns the local value. Otherwise, the symbol is resolved as a Var in the global environment and evaluating it returns the Var's root value.

If a symbol cannot be resolved, evaluation signals an error.

### 2.3.3 List evaluation

The empty list evaluates to itself. A non-empty list is evaluated according to its first element.

If the first element names a special form, the list is evaluated according to that special form. If it resolves to a macro, the macro is expanded and the resulting form is evaluated instead. Otherwise, the list is treated as a function application.

### 2.3.4 Function application

In a function application, the function target is evaluated to a callable value and the argument forms are evaluated from left to right. The callable value is then invoked with the evaluated arguments.

If the target is not callable, or if the number of arguments does not match any supported arity, a runtime error is signalled.

## 2.4 Special forms

Special forms are list forms whose evaluation rules are built into the language. Unlike ordinary function calls, they may choose which subforms are evaluated and may introduce bindings or control-flow behavior.

### 2.4.1 quote

```
(quote form)
```

Returns `form` as data without evaluating it.

For example:

```
(quote (+ 1 2))
```

returns the list `(+ 1 2)` instead of evaluating it to 3.

### 2.4.2 if

```
(if test then-expr else-expr?)
```

The question mark denotes an optional form. The `if` form first evaluates `test`. If the result is truthy, it evaluates and returns `then-expr`. Otherwise, it evaluates and returns `else-expr`, if it is present. If `else-expr` is not present and `test` evaluates to a falsy value, the result is `nil`.

Only the selected branch is evaluated. In TinyClojure, only `nil` and `false` are falsy; all other values are truthy.

For example:

```
(if nil
  1
  2)
```

evaluates to 2, while:

```
(if 0
  1
  2)
```

evaluates to 1, because 0 is truthy in TinyClojure.

### 2.4.3 do

Here, `body*` denotes zero or more body expressions.

```
(do body*)
```

Evaluates all body expressions from left to right and returns the value of the last expression. If no expressions are provided, it returns `nil`.

For example:

```
(do
  (println "first")
  (println "second")
  (+ 1 2))
```

This form prints `first`, then prints `second`, and then returns 3.

### 2.4.4 **def**

```
(def name)
; or
(def name init)
```

Creates or updates a global Var named `name`. If the Var does not exist yet, it is created with initial root value `nil`. If `init` is present, it is then evaluated and the Var root is set to the resulting value. If `init` is not present, the Var root is left unchanged.

The `def` form evaluates to the Var object itself, not to the Var's root value. Var objects and explicit dereferencing are described in Section 2.4.8.

For example:

```
(def a 5)
a

(def a)
a

(def a 3))
a
```

The first `def` creates the Var `a` and sets its root value to 5. The second `def` refers to the existing Var but does not provide a new initializer, so the root value remains 5. The third `def` updates the root value to 3. Each `def` form itself evaluates to the Var object for `a`; the following uses of `a` evaluate to the current root value.

### 2.4.5 **let\***

```
(let* (name1 value1 name2 value2 ...) body*)
```

Evaluates bindings sequentially. Each binding is visible to the following bindings and to the body. The body is evaluated left to right, and the value of the last body expression is returned.

For example:

```
(let* (x 1
      y (+ x 2))
      (println y)
      (* y 10))
```

The binding for `x` is evaluated first. The binding for `y` can therefore refer to `x`. The body then prints 3, and the value of the whole `let*` form is 30, because the last body expression is returned.

### 2.4.6 `recur`

```
(recur expr*)
```

The `recur` form transfers control to the nearest enclosing recursion point, which is either a `loop*` form or a function body. The number of expressions must match the number of bindings or parameters of the recursion target. A `recur` form is only valid in tail position. Its expressions are evaluated from left to right, and their values become the new values of the recursion target's bindings.

The next subsection shows an example of `recur` used with `loop*`.

### 2.4.7 `loop*`

```
(loop* (name1 value1 name2 value2 ...) body*)
```

Creates a recursion point with local bindings. The body is evaluated like the body of `let*`, but tail-position `recur` forms may transfer control back to the beginning of the loop with new binding values.

For example, the following loop computes the sum of numbers from 1 to 100:

```
(loop* (i 1
        acc 0)
  (if (> i 100)
    acc
    (recur (+ i 1)
            (+ acc i))))
```

The loop starts with `i` bound to 1 and `acc` bound to 0. Each `recur` evaluates the new values of `i` and `acc` and restarts the loop body with these updated bindings.

### 2.4.8 `var`

```
(var name)
```

Returns the Var object associated with `name` without dereferencing it. The argument must be a symbol.

Normally, evaluating a symbol that resolves to a Var returns the Var's root value. The `var` form returns the Var object itself instead. The root value can be obtained explicitly with the built-in function `deref`.

```
(def x 42)

x           ; evaluates to 42
(var x)     ; evaluates to the Var object for x
(deref (var x)) ; evaluates to 42
```

Here, `x` performs the ordinary symbol lookup and dereferences the Var implicitly. The form `(var x)` returns the Var object itself, while `(deref (var x))` explicitly reads its current root value.

### 2.4.9 `fn*`

The `fn*` form creates a function value. Its detailed syntax is described in the following section.

## 2.5 Functions

Functions are first-class values. A function can be stored in a Var, passed as an argument, returned from another function, and invoked. TinyClojure functions are created using the `fn*` special form.

### 2.5.1 Basic function syntax

A basic function (i.e. a function with only 1 arity) has the following form:

```
(fn* (params*) body*)
```

The parameter list introduces local bindings visible in the function body. When the function is invoked, the arguments are bound to the corresponding parameters and the body is evaluated from left to right. The value of the last body expression becomes the return value of the function. If the body is empty, the function returns `nil`.

For example:

```
(def add-one
  (fn* (x)
    (println "x =" x)
    (+ x 1)))

(add-one 4)
```

The function `add-one` binds its argument to `x`. Its body contains two forms: the first prints the value of `x` to the standard output, and the second computes the return value. As in `do` and `let*`, the value of the last body form is returned, so `(add-one 4)` evaluates to 5.

A function may also be given an internal name hint:

```
(fn* name (params*) body*)
```

In the current version of TinyClojure, `name` is used only as a hint for the generated internal function/module name. It does not introduce a local binding in the function body. Unlike named functions in Clojure, this form therefore cannot be used for direct self-reference.

### 2.5.2 Multiple arities

A function may define several arities:

```
(fn*  
  ((x) body1*)  
  ((x y) body2*))
```

When the function is invoked, the implementation selects the overload whose parameter count matches the number of arguments. If no overload matches, invocation signals an arity error. Two fixed arities with the same number of parameters are not allowed.

For example:

```
(def describe  
  (fn*  
    (() "no arguments")  
    ((x) "one argument")  
    ((x y) "two arguments")))  
  
(describe)  
(describe 1)  
(describe 1 2)
```

Each call selects the overload whose parameter count matches the number of supplied arguments.

A `recur` form inside a function body refers to the current overload, not to the function as a whole. For example:

```
(def countdown  
  (fn*  
    ((n)  
      (if (= n 0)  
          nil  
          (recur (- n 1)))))  
    ((n message)  
      (if (= n 0)  
          message  
          (recur (- n 1) message)))))
```

The first `recur` targets the one-parameter overload, while the second `recur` targets the two-parameter overload. A `recur` in the first overload cannot jump

to the second overload; calling another overload requires an ordinary function call.

This has two consequences:

- The number of `recur` arguments must match the current overload's parameter count; otherwise, the program is rejected during compilation.
- The `recur` form transfers control directly to the beginning of the current overload, bypassing the function's arity-dispatching entry point.

A single-arity function may also be written using the multiple-arity form. Therefore, the following two definitions are equivalent:

```
(defn foo1
  (x y)
  (+ x y))

(defn foo2
  ((x y)
   (+ x y)))
```

### 2.5.3 Variadic functions

A function may define a variadic arity using `'&'` in the parameter list:

```
(fn* (x y & rest) body*)
```

The parameters before `'&'` are required. Any remaining arguments are collected into a list and bound to the parameter after `'&'`. If no extra arguments are provided, the rest parameter is bound to `nil`.

For example:

```
(def collect
  (fn* (x & rest)
    rest))

(collect 1)
(collect 1 2 3)
```

The first call binds `rest` to `nil`, while the second call binds `rest` to the list `(2 3)`.

A function may contain at most one variadic arity. A variadic arity with `n` required parameters can handle calls with `n` or more arguments. If the same function also defines a fixed arity with exactly `n` parameters, calls with exactly `n` arguments use the fixed arity, while calls with more than `n` arguments use the variadic arity.

For example:

```
(def func
  (fn*
    ((a) (println "A"))
    ((a b) (println "B"))
    ((a b & args) (println "C"))))

(func 1 2) ; prints "B"
(func 1 2 3) ; prints "C"
```

Fixed arities with more than `n` parameters conflict with such a variadic arity, because calls with that many arguments would be ambiguous. Such function definitions are rejected.

Within a variadic overload, `recur` must provide one value for each required parameter and one value for the rest parameter. The last parameter of `recur` becomes the new value of the rest parameter directly; it is not expanded, collected, or otherwise transformed.

### 2.5.4 Closures

A function may refer to local bindings from an enclosing function or `let*`. Such bindings are captured by the function. The captured values remain available when the function is invoked, even if the original scope has already returned.

For example:

```
(def make-adder
  (fn* (x)
    (fn* (y)
      (+ x y))))

(def add-five (make-adder 5))

(add-five 3)
```

The inner function captures `x`. The call `(add-five 3)` therefore evaluates `(+ 5 3)`.

### 2.5.5 Function invocation

When a function is invoked, the number of supplied arguments is compared with its supported arities. If a matching arity is found, the corresponding body is evaluated in a new local environment. If no matching arity exists, a runtime error is signalled.

## 2.6 Macros

Macros are functions executed during macroexpansion. They transform source forms before those forms are semantically analyzed and compiled.

TinyClojure macros are defined using `defmacro`:

```
(defmacro name (params*) body*)
```

The parameter list follows the same arity rules as ordinary functions. However, when a macro is invoked, its arguments are not evaluated before the call. Instead, the raw argument forms are passed to the macro. The macro body is evaluated during macroexpansion, and its result must be a valid TinyClojure form.

For example, a simple macro can expand into an `if` form:

```
(defmacro when (test & body)
  `(if ~test
      (do ~@body)))
```

A use of the macro:

```
(when condition
  (println "true"))
```

is expanded before semantic analysis into a form equivalent to:

```
(if condition
  (do (println "true")))
```

Macroexpansion is recursive: after a macro invocation is expanded, the resulting form is processed again, so it may itself contain macro calls.

TinyClojure implements only a simplified macro system. It does not aim to support all features of Clojure macros, such as namespace-qualified symbol resolution in syntax quote, metadata handling, or access to special macro-only values such as `&form`, which contains the form currently being macroexpanded, and `&env`, which contains the current lexical environment.

## 2.7 Modules and top-level forms

A TinyClojure source file is treated as a module. A module consists of a sequence of top-level forms, which are processed from left to right. Top-level forms may define Vars, define macros, or perform ordinary computation for side effects.

The most common top-level forms are `def`, `defmacro`, and function definitions expressed using `def` together with `fn*`. A top-level `def` creates or updates a Var in the global environment. A top-level `defmacro` creates a macro that

may be used by later forms in the same module or by modules loaded afterwards.

TinyClojure does not implement Clojure's full namespace system. In Clojure, namespace groups Vars and is usually declared using the `ns` form. The `ns` form also supports features such as requiring other namespaces, creating aliases, and referring to definitions from other namespaces. TinyClojure does not implement this form. Instead, modules are loaded explicitly by the implementation, as described in later chapters.

When a module is loaded, its dependencies must be loaded before its own top-level forms are executed. Loading an already loaded module does not repeat its top-level execution.

For example, a source file `math.clj` may contain:

```
(def pi 3.1415)

(defn square (x)
  (* x x))

(defn sum-of-squares (a b)
  (+ (square a) (square b)))

(def circle-area (rad)
  (* pi r r))

(println (sum-of-squares 3 4))
```

This file is treated as one module. Loading the module processes the forms from left to right. The first form creates a global Var `pi` whose root is the number `3.14159`. The next 3 forms create Vars whose root values are function objects. The final form is an ordinary top-level expression; it is evaluated for its side effect and prints the result.

## 2.8 Errors and limitations

TinyClojure is dynamically typed, so some invalid programs are detected only during execution. Errors may occur while reading source code, compiling forms, expanding macros, or executing generated code. The language does not define a structured exception system of its own; errors are reported by the implementation.

The current implementation reports these errors using exceptions provided by the underlying host language, C++.

TinyClojure does not provide user-level exception handling. In the REPL, runtime errors are caught and reported to the user, after which the REPL

continues. Outside the REPL, error handling depends on the C++ API used to execute the program.

The implemented subset intentionally omits many features of full Clojure, including persistent vectors, maps, sets, lazy sequences, protocols, multi-methods, metadata, destructuring, platform interoperability, concurrency primitives, and the full namespace system.

Most common runtime errors, such as division by zero, unsupported arities, calling non-callable values, or passing values of incorrect types to built-in functions, are explicitly checked. Some low-level implementation limitations remain. In particular, not all arithmetic overflow cases are checked, so programs that rely on signed integer overflow are outside the supported subset.

## 2.9 Core library

TinyClojure provides a small core library loaded during runtime initialization. The core library is not intended to be a complete replacement for `clojure.core`; it only contains the functions and macros needed by the implemented subset and by the examples used in this thesis.

The runtime first registers a set of native built-in functions implemented in C++. These functions provide primitive operations such as arithmetic, comparison, printing, list manipulation, type predicates, reading and evaluating forms, macroexpansion, and module loading. The Clojure-level core library is then compiled and loaded on top of these built-ins.

Important groups of core functions include:

- arithmetic and comparison: `+`, `-`, `*`, `/`, `<`, `<=`, `>`, `>=`, `=`, `not=`;
- list and sequence operations: `list`, `cons`, `first`, `next`, `seq`, `list*`, `concat`, `count`, `map`, `filter`, `reduce`, `reverse`;
- predicates and conversion functions: `nil?`, `string?`, `symbol?`, `list?`, `function?`, `integer?`, `double?`, `boolean?`, `var?`, `character?`, `str`, `double`, `long`;
- higher-order helpers: `identity`, `apply`, `comp`, `partial`;
- input, output, and utility functions: `print`, `println`, `read`, `read-string`, `slurp`, `spit`, `gensym`, `epoch-nanos`;
- macro-related functions: `set-macro!`, `macroexpand`, `macroexpand1`;
- module-related functions: `compile-module`, `load-module`.
- var operations: `deref`, `var?`

The core library also defines several macros, including `fn`, `defn`, `defmacro`, `let`, `loop`, `when`, `when-not`, `if-not`, `cond`, `and`, `or`, `time`, `bench`, `dotimes`, `if-let`, `when-let`, `->`, and `->>`.

Only selected functions are described in detail in this thesis when they are relevant to the compiler or runtime design. The exact implementation of the core library is included in the source attachment.

Several familiar Clojure-like forms are not primitive in TinyClojure. For example, `fn`, `defn`, `let`, and `loop` are macros defined in the core library and expand to the primitive forms `fn*`, `def`, `let*`, and `loop*`.

## Runtime system

The previous chapter defined TinyClojure as a dynamically typed Lisp-like language with lists, Vars, first-class functions, closures, macros, and module loading. These features are part of the language model, but they are not provided directly by LLVM. LLVM IR provides low-level functions, basic blocks, instructions, and memory operations, but it does not define a managed object model, dynamic dispatch, garbage collection, global Vars, or Clojure-like function objects.

For this reason, TinyClojure is implemented together with a runtime system written in C++. This matches the implementation language of the compiler and integrates naturally with the LLVM C++ API used by the project. Internally, the runtime can use C++ data structures, while generated LLVM IR interacts with it through a small set of C-compatible helper functions.

Generated LLVM IR does not operate directly on raw integers, strings, lists, or functions in most situations. Instead, it manipulates pointers to runtime objects. Operations that require dynamic behavior, such as arithmetic on boxed values, list manipulation, function invocation, Var dereferencing, module loading, or memory allocation, are implemented as calls to runtime functions.

The runtime system has four main responsibilities. First, it defines a uniform representation of TinyClojure values. Second, it provides primitive operations used by generated code and by the core library. Third, it manages global state, including Vars, loaded modules, and runtime initialization. Finally, it provides memory management for heap-allocated runtime objects.

This chapter describes the main parts of the TinyClojure runtime. It first introduces the uniform object representation and the concrete runtime value types. It then describes function objects and the uniform calling convention used by both user-defined and built-in functions. The following sections describe global Vars, module loading, and memory management.

### 3.1 Object representation

TinyClojure is dynamically typed, so the concrete type of a value is generally known only at runtime. The current implementation does not perform static type inference or specialization. Generated code therefore cannot usually assume whether an expression evaluates to an integer, string, list, function, or another kind of value.

For this reason, most TinyClojure values use a uniform boxed representation. Generated LLVM IR passes values around as pointers to runtime objects. The runtime object stores a type tag, a pointer to type-specific payload data, a pointer to a method table, and metadata used by the garbage collector.

A simplified version of the runtime object structure is shown in Listing 5.

```
struct Object {  
    void *m_Data;  
    ObjectType m_Type;  
    const MethodTable *m_MethodTable;  
    mutable bool m_Marked = false;  
    bool m_Static = false;  
};
```

■ **Listing 5** Simplified TinyClojure runtime object representation.

The fields have the following roles:

- `m_Data` points to type-specific payload data. For example, an integer object points to an integer payload, while a list object points to a list cell.
- `m_Type` stores the broad runtime type tag, such as integer, string, list, Var, function, or closure. Runtime helpers use this tag to check supported operations and report type errors.
- `m_MethodTable` points to operations associated with the object. This is used for value-specific behavior such as invocation and textual conversion.
- `m_Marked` and `m_Static` are used by the memory manager. They are described later in Section 3.6.

The same structure can represent many different runtime values. For example, an integer and a function object both have the same outer `Object` wrapper, but their payloads and method tables differ.

#### 3.1.1 Method tables

The method table stores native operations associated with a runtime object. Its structure is shown in Listing 6.

```
struct MethodTable {
    CallFn m_CallFn;
    EqualsFn m_EqualsFn;
    UnaryFn m_ToStringFn;
    UnaryFn m_ToEdnFn;
};
```

■ **Listing 6** Structure of the TinyClojure method table.

The `m_CallFn` entry is used for callable values. For non-callable values, such as integers or strings, this entry is `nullptr`. The `m_EqualsFn` entry is used for object comparison. The textual conversion entries `m_ToStringFn` and `m_ToEdnFn` are used by printing and debugging support.

The relationship between the uniform `Object` wrapper, the type-specific payload, and the method table is illustrated in Table 1. The examples are schematic, but they reflect the important pointer relationships: the runtime object stores a pointer to type-specific payload data and a pointer to a method table.

| Runtime object                                                            | Payload pointed to by <code>m_Data</code>                 | Static method table pointed to by <code>m_MethodTable</code>                                                                                                                                                                       |
|---------------------------------------------------------------------------|-----------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Integer object<br><code>m_Type = INTEGER</code>                           | <code>TCInteger</code><br><code>m_Value = 42</code>       | <code>TCInteger::st_MethodTable</code><br><code>m_CallFn = nullptr</code><br><code>m_EqualsFn = TCInteger::equals</code><br><code>m_ToStringFn = TCInteger::toString</code><br><code>m_ToEdnFn = TCInteger::toEDN</code>           |
| Function object for <code>concat</code><br><code>m_Type = FUNCTION</code> | <code>TCFunction</code><br><code>m_Name = "concat"</code> | generated static method table<br><code>m_CallFn</code> → generated call entry for <code>concat</code><br><code>m_EqualsFn = nullptr</code><br><code>m_ToStringFn = TCFunction::toString</code><br><code>m_ToEdnFn = nullptr</code> |

■ **Table 1** Examples of runtime objects, their payloads, and their static method tables.

For ordinary value types such as integers, all objects of the same runtime type can share one static method table. Function objects use the same outer `Object` representation, but their method table can be specific to the compiled function. Its `m_CallFn` entry points to the generated invocation function, where arity dispatch is implemented. The `TCFunction` payload itself stores only function-level runtime data such as the function name.

### 3.1.2 Runtime helper functions

Generated LLVM IR usually does not manipulate runtime payloads directly. Instead, it calls C-compatible runtime helper functions implemented in C++. These helpers allocate runtime objects, perform type checks, implement dynamic operations such as equality and invocation, and report runtime errors.

For example, equality is implemented by the runtime helper `tc_object_equals`. The helper first handles trivial cases, such as pointer equality and mismatched runtime types. If both operands have the same runtime type, it uses the object's method table to find a type-specific equality implementation.

```
const Object *tc_object_equals(const Object *obj1,
                              const Object *obj2) {
    if (obj1 == obj2) {
        return tc_boolean_const_true;
    }

    if (!obj1 || !obj2 || obj1->m_Type != obj2->m_Type) {
        return tc_boolean_const_false;
    }

    EqualsFn equalsFn = obj1->m_MethodTable->m_EqualsFn;
    if (equalsFn) {
        return equalsFn(obj1, obj2);
    }

    return tc_boolean_const_false;
}
```

■ **Listing 7** Simplified runtime helper for TinyClojure equality.

Listing 7 illustrates the role of the method table in dynamic operations. The generated code can call one generic helper for equality, while the helper dispatches to a type-specific implementation when one is available. Integer equality can therefore compare integer payload values, string equality can compare string contents, and objects without a custom equality function are equal only when they were already caught by the initial pointer-equality check.

Generated LLVM IR usually creates runtime values by calling runtime helper functions. For example, helpers such as `tc_integer_new`, `tc_double_new`, `tc_string_new`, and `tc_list_cons` allocate and initialize boxed runtime objects. Other helpers, such as `tc_object_equals`, implement dynamic operations over already constructed objects.

This means that generated code does not need to know the concrete C++ payload layout of every runtime value. It can request a new boxed integer, string, or list cell through a helper function and then continue working with the returned `Object*`.

### 3.1.3 Why the method table is stored in each object

For simple runtime types, such as integers or strings, all values of the same type can share the same method table. In that case, the method table could theoretically be obtained from a global table indexed by `m_Type`.

However, this would not be sufficient for all callable values. TinyClojure does not assign a distinct runtime type to every compiled function. Many functions have the same broad runtime type, but different invocation behavior. They may have different generated entry points and arity-dispatch logic. Closures additionally store captured values in their payload. The type tag alone therefore cannot determine how a particular callable object should be invoked.

For this reason, the method table pointer is stored directly in the object. This allows different function objects to have different invocation entries while still sharing the same uniform `Object` representation.

This design also works well with the runtime helper interface described above. Generated LLVM IR can operate mostly on `Object*` values and call helper functions when it needs type-specific behavior.

## 3.2 Runtime value types

The `Object` structure provides a uniform wrapper, but each runtime type has its own payload representation. This section summarizes the value types used by the runtime and explains how they relate to the language values defined in Section 2.2.

The value `nil` is represented as a null object pointer. This makes checks for `nil` simple in both generated code and runtime helper functions. Since only `nil` and `false` are falsy, conditional expressions can be compiled by comparing a value with the null pointer and with the singleton object representing `false`.

Boolean values are represented by two static singleton objects, one for `true` and one for `false`. Since booleans are immutable and only two boolean values exist, allocating a new object for every boolean result would be unnecessary.

Integer and floating-point values are represented as boxed runtime objects. An integer object stores a signed 64-bit integer payload, while a floating-point object stores a 64-bit double payload. Arithmetic functions inspect the

runtime types of their arguments and either perform the operation or report a runtime error.

To reduce allocation pressure, the runtime caches integer objects for a configurable range of small values. When an integer in this range is requested, the runtime returns the existing cached object instead of allocating a new one. This optimization is similar to the caching performed by Java's `Integer.valueOf`, which is specified to always cache values in the range `[-128, 127]` and may cache additional values outside this range [19]. TinyClojure uses the same range by default, but allows the range to be configured using the application's command-line arguments.

Small-integer caching is also used by other language runtimes. For example, CPython preallocates small integer objects so that they can be shared; in CPython 3.12, the preallocated range is `[-5, 256]` [20]. This is an implementation detail of CPython rather than a semantic property of the Python language, but it illustrates the same general optimization.

Characters, strings, and symbols are also represented as runtime objects. Symbols are especially important because source forms are first read as data: before semantic analysis, a symbol such as `x` is simply a runtime value that may later be resolved as a local binding, a `Var`, a special form, or a macro name.

Lists are represented using list cells. A list cell stores one value and a reference to the rest of the list. This representation supports operations such as `first`, `next`, and `cons`, and also fits the representation of source forms, since a compound form such as `(if test then else)` is read as a list.

Vars and functions are runtime values too, but they have additional behavior. Vars are mutable global references, while functions are callable objects. They are therefore described in separate sections.

### 3.3 Function representation and calling convention

Functions are first-class values in TinyClojure, so they are represented as runtime objects. A function value can be stored in a `Var`, passed as an argument, returned from another function, captured by a closure, and invoked later.

The runtime therefore represents functions using the same `Object` wrapper as other values. Unlike non-callable values such as integers or strings, function objects provide an invocation operation through their method table.

### 3.3.1 Uniform calling convention

Generated LLVM code invokes callable values through a uniform calling convention. Conceptually, every callable object is called with three pieces of information: the callable object itself, the number of supplied arguments, and an array containing the argument values.

The callable object is passed as the first argument so that closures can access their captured environment. The argument count and argument array make it possible to support multiple arities and variadic functions without requiring a separate low-level calling convention for each arity.

This convention is used both for user-defined functions and for native built-in functions implemented in C++. As a result, generated code does not need to distinguish between a function compiled from TinyClojure source code and a built-in runtime function. Both are represented as callable runtime objects.

Conceptually, every callable entry point has the following shape:

```
Object *invoke(Object *self, unsigned argc, Object **argv)
```

Here, `self` is the callable object being invoked, `argc` is the number of supplied arguments, and `argv` points to the argument values. All arguments and return values are represented as `Object*`.

### 3.3.2 Arity dispatch

A TinyClojure function may contain several overloads, each corresponding to one supported arity. The runtime-visible function object does not directly represent only one body. Instead, invocation first enters a dispatch function. The dispatch function examines the number of supplied arguments and transfers control to the matching overload.

A TinyClojure function may contain several overloads, each corresponding to one supported arity case. The runtime-visible function object does not store these overloads directly. Instead, its method table points to a generated call entry. This entry receives the function object, the number of supplied arguments, and the argument array according to the uniform calling convention.

The generated call entry implements arity dispatch. It examines `argc` and transfers control to the matching overload body. If no matching fixed-arity or variadic overload exists, it reports an arity error.

For a function with two fixed-arity overloads, the generated dispatch logic can be viewed schematically as follows:

```

if argc == 1:
    call overload for one argument
else if argc == 2:
    call overload for two arguments
else:
    report arity error

```

■ **Listing 8** Schematic arity dispatch for fixed-arity overloads.

A variadic overload is handled by checking whether the supplied argument count is at least the number of required parameters. If the check succeeds, the remaining arguments are collected into a list and passed as the variadic parameter.

For example, consider a function with one fixed-arity overload accepting two arguments and one variadic overload with two required parameters:

```

fixed arity 2      -> selected when argc == 2
variadic arity 2+  -> selected when argc > 2
otherwise          -> arity error

```

■ **Listing 9** Schematic arity dispatch with a variadic overload.

A call with two arguments selects the fixed-arity overload. A call with more than two arguments selects the variadic overload and passes the additional arguments as a list. A call with fewer than two arguments reports an arity error.

### 3.3.3 Built-in functions

Built-in functions use the same callable-object interface as functions compiled from TinyClojure source code. Their invocation operation points to a native C++ function rather than to generated LLVM code.

For example, a built-in addition function can use the same entry-point shape as a compiled TinyClojure function:

```
Object* tinyclj_rt_add(Object* self, unsigned argc, Object** argv)
```

The implementation of the built-in function is native C++ code, but generated LLVM IR can invoke it through the same callable-object interface as any other function.

This design keeps the boundary between generated code and the runtime simple. The compiler only needs to emit calls following one calling convention, while the runtime can provide different callable objects behind that interface.

### 3.3.4 Closures

A closure is a function value that carries values captured from its surrounding lexical scope. In TinyClosures, closures use the same outer `Object` representation as other functions, but their payload additionally stores the captured values. Captured values are not passed as ordinary call arguments. They are stored in the function object when the closure is created.

When a function expression is evaluated, the generated code creates a runtime function object. If the function captures variables from an enclosing lexical scope, the current values of those variables are stored in the closure payload. The method table of the object still provides the call operation, while the payload provides the captured environment needed by the function body.

Schematically, a closure object can be viewed as follows:

```
closure Object
  m_Type      = FUNCTION
  m_MethodTable -> generated method table
                  m_CallFn -> generated call entry
  m_Data      -> closure payload
                  captured[0] -> first captured value
                  captured[1] -> second captured value
```

■ **Listing 10** Schematic representation of a closure object.

When the closure is invoked, the generated call entry receives the closure object as the `self` argument of the uniform calling convention. It can then load captured values from the closure payload and use them while evaluating the function body. The ordinary argument array is reserved for arguments supplied by the caller; captured values are obtained through `self`.

### 3.3.5 Dynamic invocation and direct linking

The general invocation path is dynamic. Generated code evaluates the call target, obtains a runtime value, checks that the value is callable, and invokes it through the object's method table. This path is necessary for higher-order functions, closures, and calls whose target is not known until runtime.

For example, in the following higher-order function, the target of the call `(f x)` is not known statically:

```
(def apply-once
  (fn* (f x)
    (f x)))
```

The generated code must therefore evaluate `f`, check that the resulting value is callable, and invoke it through the method table.

Calls through global Vars also normally use the dynamic path. A symbol referring to a Var is evaluated by dereferencing the Var to obtain its current root value. If that value is callable, it is then invoked through the method table. This preserves the dynamic behavior of Vars: if a Var is later redefined, subsequent calls through the Var observe the new value.

When the compiler can determine the target function during semantic analysis, it may use direct linking. In this case, the generated code calls a known generated function entry point directly, instead of dereferencing the Var and invoking the resulting value through the method table. This reduces call overhead, but it also means that later redefinitions of the Var are not observed by that direct call.

For example:

```
(def add-one
  (fn* (x)
    (+ x 1)))

(add-one 4)
```

If `add-one` is resolved to a known function at compile time, the call may be compiled as a direct call to the generated entry point for `add-one`.

This optimization is similar in purpose to direct linking in JVM Clojure, discussed in Section 1.5. TinyClojure implements direct linking for calls whose target has been resolved to a known function object during semantic analysis.

Each callable object stores enough information for the compiler to identify its native entry point. When direct linking is used, generated code calls this entry point directly instead of dereferencing the Var at runtime and invoking the resulting value through the method table.

This avoids the ordinary dynamic invocation path. The trade-off is that later redefinitions of the Var are not observed by that direct call. Functions are still represented as ordinary callable objects, because they may also be stored in Vars, passed as arguments, returned from functions, and captured by closures.

### 3.4 Vars and global environment

TinyClojure uses Vars to represent global definitions at runtime. A Var is a mutable reference object stored in the global environment. The global environment maps names to Var objects, not directly to the values stored in those Vars.

This indirection is important because global definitions may be updated. Generated code can resolve a global name to a Var and then obtain the Var's

current root value by dereferencing it. If the Var is later rebound, ordinary Var dereferencing observes the new value.

### 3.4.1 Vars

A Var stores a name and a root value. The root value is the current value associated with the global definition. For example, after evaluating a top-level definition such as:

```
(def x 42)
```

the global environment contains a Var named `x`, and the root value of that Var is the integer value `42`.

When a symbol resolves to a global Var during ordinary expression evaluation, the Var is dereferenced and the root value is used. The Var object itself is obtained only when the program explicitly uses the `var` special form, as described in Section 2.4.8.

### 3.4.2 Global environment

The global environment is owned by the runtime. Generated LLVM IR does not access its internal data structure directly. Instead, it calls runtime helper functions to create or find Vars, update root values, dereference Vars, and look up global definitions.

This keeps the concrete representation of the global environment inside the C++ runtime. The generated code depends only on the runtime interface, not on the specific container used to map names to Vars.

Because the global environment stores Var objects rather than plain values, rebinding a global definition updates the root value stored in an existing Var. The mapping from the name to the Var itself remains stable.

### 3.4.3 Top-level definitions

Top-level definitions are executed as part of module initialization. A `def` form creates the corresponding Var if it does not already exist. If the form contains an initializer, the initializer is evaluated and the Var's root value is updated.

Function definitions are represented using the same mechanism. A `defn` form is expanded into a definition whose value is a function object. The function name is therefore a global symbol that resolves to a Var whose root value is callable.

This also means that redefining a function updates the root value of its Var. Calls that go through ordinary Var dereferencing observe the updated value.

Calls compiled using direct linking may bypass this indirection, as described in Section 3.3.5.

### 3.4.4 Macros and Vars

Macros are stored in Vars as well. A macro definition creates a function object and marks the corresponding Var as a macro Var. During macroexpansion, if the first element of a list resolves to a macro Var, the macro function is invoked with the unevaluated argument forms. The returned form replaces the original form and is expanded and analyzed instead.

Using Vars for macros makes macro definitions part of the same global environment as ordinary definitions. The difference is not the storage mechanism, but the way the compiler treats Vars marked as macros during macroexpansion.

## 3.5 Module loading support

TinyClojure source files are compiled and loaded as modules. The language-level behavior of modules was described in Section 2.7: a module contains top-level forms that are processed from left to right, and dependencies must be loaded before the module itself. The runtime provides the state and helper functions needed to implement this behavior.

### 3.5.1 Runtime module state

The runtime identifies modules by their module names. A module name is derived from the source file path relative to the source root. For example, a source file such as `core.clj` is represented by the module name `core`, while a file in a subdirectory can be represented by a compound module name derived from its relative path.

The runtime stores the names of modules that have already been loaded. Before loading a module, it checks whether the module name is already present in this set. If it is, the load request returns without executing the module again. Otherwise, the module's dependencies are loaded first, and the module's top-level initialization code is executed. After successful initialization, the module name is recorded as loaded.

The runtime also tracks modules that are currently being loaded. This allows it to detect recursive load requests and avoid starting the same module initialization twice before the first initialization has finished.

This prevents repeated execution of top-level forms when the same module is requested multiple times. This is important because top-level forms may

have side effects, such as binding Vars, defining macros, printing values, or loading additional modules.

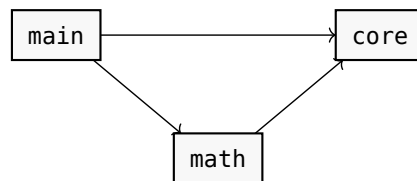
Loaded-module tracking also handles shared dependencies. If two modules depend on the same third module, the dependency is initialized only once.

### 3.5.2 Loading dependencies

Before a module's own top-level code is executed, its dependencies must be loaded. The runtime module loader therefore first checks whether each dependency has already been loaded. If not, it recursively loads that dependency.

This gives module loading a simple dependency-ordering model: dependency modules initialize their Vars and macros before the module that depends on them is evaluated.

For example, suppose module `main` depends on both `math` and `core`, and module `math` also depends on `core`. This dependency graph is shown in Figure 1.



■ **Figure 1** Example module dependency graph. Arrows point from a module to a module it depends on. Loading `main` requires both `math` and `core`, but `core` is initialized only once even though it is reachable through two dependency paths.

A request to load `main` recursively loads its dependencies before executing `main`'s own top-level code. The module `core` is reachable both directly from `main` and indirectly through `math`, but the runtime records loaded modules by name, so `core` is initialized only once.

The arrows in Figure 1 describe dependency relationships, not initialization order. Initialization proceeds from dependencies to dependents.

### 3.5.3 Executing top-level code

A compiled module contains code corresponding to its top-level forms. During loading, this code is executed to perform the module's initialization. Typical top-level actions include declaring Vars, binding root values, defining functions, marking macro Vars, loading additional modules, and evaluating ordinary top-level expressions such as calls to `println`.

The runtime does not need to understand the meaning of every top-level form. That meaning has already been determined by the compiler. The runtime only provides the operations needed by the generated initialization code. For this reason, loading a module may have observable effects even if no function from that module is called afterwards.

### 3.5.4 Running and loading modules

TinyClojure supports both JIT execution and ahead-of-time (AOT) compiled modules. In the JIT case, LLVM modules are generated and added to the running execution engine during the current session. In the AOT case, module artifacts are produced ahead of time and loaded later by the runtime.

The current implementation does not produce a standalone native executable for a TinyClojure program. Even when a module is compiled ahead of time, it still has to be loaded by the TinyClojure runtime. In practice, the user starts the TinyClojure executable or REPL and explicitly loads the module.

AOT compilation is useful when the same module is loaded in a later runtime session. The first session compiles the module and writes loadable artifacts:

```
user=> (compile-module "main")
nil

user=> exit
```

In a later session, the user can load the already compiled module instead of compiling it from source again:

```
user=> (load-module "main")
nil

user=> (sum-of-squares 3 4)
25
```

The important distinction is that `compile-module` produces persistent AOT artifacts, while `load-module` initializes a module in the current runtime session. During `load-module`, the runtime loads dependencies, links the generated code with runtime helpers, and executes the module initialization code. After loading finishes, Vars defined by the loaded modules are available in the global environment and can be used by later REPL forms.

The runtime also checks whether an AOT artifact is still up to date with respect to the corresponding source file. If the source file has been modified after the artifact was produced, `load-module` treats the compiled artifact as stale and loads the module from source instead. The stale artifact is not updated automatically; the user must explicitly call `compile-module` again to

produce a new AOT version. This makes AOT compilation an explicit caching mechanism rather than an automatic build system.

## 3.6 Memory management

TinyClojure values are represented as runtime objects, and most of these objects are allocated on the heap. This includes boxed numbers, strings, symbols, list cells, Vars, function objects, and closure payloads. In the current implementation, the compiler does not perform lifetime analysis, ownership analysis, or escape analysis. As a result, it generally cannot determine statically when a heap object can be freed. For example, a list may be returned from a function, stored in a Var, or captured by a closure.

Manual memory management would make the language difficult to use and would not match the programming model of Clojure-like languages. The TinyClojure runtime therefore includes a simple garbage collector responsible for reclaiming unreachable runtime objects.

The implemented collector is a simple mark-and-sweep collector [21]. During collection, the runtime first marks objects reachable from a set of known roots. It then scans the objects known to the memory manager and frees those that were not marked.

### 3.6.1 Allocation

Runtime objects are allocated through runtime helper functions rather than directly by generated LLVM code. For example, generated code calls helper functions to create integers, strings, symbols, lists, Vars, and function objects. This keeps allocation centralized in the runtime.

Objects are allocated using ordinary C++ allocation. The memory manager does not store objects directly inside its internal collection. Instead, it stores pointers to allocated objects in a tracking vector. This vector is used during collection to iterate over objects known to the garbage collector.

Centralized allocation has two advantages. First, the runtime can track allocated objects so that they can later be considered by the garbage collector. Second, allocation can apply runtime-specific optimizations, such as reusing static singleton objects or cached integer objects instead of allocating a new object.

Not every value is allocated in the same way. The value `nil` is represented as a null object pointer. Boolean values are represented by static singleton objects. Cached small integers are also treated as long-lived static objects. Other runtime values are allocated on the heap and may be reclaimed when they become unreachable.

### 3.6.2 Root set

The garbage collector starts from a root set maintained by the runtime. The roots are the global environment, which stores global Vars, and statically allocated runtime objects such as cached integers and boolean singletons. A Var itself is reachable through the global environment, and the value stored in its root binding must also remain alive.

The global environment described in Section 3.4.2 is part of the root set. The collector treats Vars stored in the global environment as reachable and follows each Var to its current root value. If a Var is rebound, the old root value remains live only if it is reachable from somewhere else.

Reachability is then followed through runtime objects. For example, a list cell references its first element and the rest of the list, a Var references its root value, and a closure references its captured values. These references are traversed during the mark phase.

A schematic example is shown in Listing 11.

```

(defn make-adder (x)
  (fn* (y)
    (+ x y)))

(def adder (make-adder 2))

(println (adder 1000))

(def adder (make-adder 5))

```

```

Root set
├ global environment
│   ├── Var make-adder → function object
│   └── Var adder      → closure C2
│                           └ captured env → cached integer 5
└ static objects
    ├── boolean singletons
    └ cached integers ..., 2, 5, ...

Unreachable after the final form
└ closure C1 from (make-adder 2)
    └ captured env → cached integer 2

Temporaries from (println (adder 1000))
├ integer object 1000
└ result of (+ 2 1000)

```

■ **Listing 11** Example of TinyClosure roots and reachability after a sequence of top-level forms. The diagram shows the state after the final `def` form has finished.

In the example, the global Var `adder` is first bound to a closure created by `(make-adder 2)`. Later, it is rebound to a closure created by `(make-adder 5)`. After the final top-level form finishes, only the newer closure remains reachable through the global environment. The older closure and its captured environment are no longer reachable and may therefore be reclaimed during sweeping.

The captured integers 2 and 5 are part of the small-integer cache and remain available as static runtime objects. Therefore, rebinding `adder` makes the old closure unreachable, but it does not reclaim the cached integer 2 itself.

The call `(println (adder 1000))` illustrates a different case. The value 1000 is outside the small-integer cache and is represented as an ordinary runtime object. The result of the addition is also temporary unless it is stored somewhere. These values are needed only while that top-level form is being evaluated and printed.

### 3.6.3 Collection points

The current implementation does not precisely track all temporary values created while generated code or runtime helper functions are executing. For this reason, garbage collection is performed only at safe points.

In the REPL, the runtime checks whether collection is needed only after a top-level form has been evaluated, its result has been printed, and before the prompt is returned to the user. At that point, evaluation-time temporaries are no longer needed unless they have been stored in a reachable runtime object, such as a Var, list, or closure.

This design keeps generated user functions and built-in runtime functions simple. They can focus on their ordinary computation logic and do not need to register every local temporary value as a garbage collector root. The trade-off is that allocation-heavy code may retain temporary objects until the current top-level evaluation finishes.

### 3.6.4 Mark phase

The mark phase starts from the root set and recursively visits objects reachable from it. Each visited heap object is marked. If the object contains references to other runtime objects, these references are marked as well.

The core marking procedure can be represented schematically as follows:

```
mark(obj):
    if obj is null:
        return
    if obj is static:
        return
    if obj is already marked:
        return

    obj.marked = true

    switch obj.type:
    case LIST:
        mark(obj.head)
        mark(obj.tail)

    case VAR:
        mark(obj.root)

    case CLOSURE:
        for value in obj.captures:
            mark(value)

    default:
        return
```

The exact implementation is written in C++, but the structure follows this procedure. Atomic values such as integers, doubles, characters, and strings do not contain references to other TinyClojure objects, so marking them only sets the mark bit. Compound values such as lists, Vars, and closures recursively mark the objects they reference.

### 3.6.5 Sweep phase

After marking is complete, the collector scans the tracking vector containing pointers to allocated objects. Objects that were marked are still reachable, so their mark flag is cleared in preparation for the next collection. Objects that were not marked are unreachable and can be destroyed.

Objects marked as static are skipped during sweeping. They are owned by the runtime for the duration of the program and are used for values such as boolean singletons and cached integers.

When an unreachable object is destroyed, its pointer is removed from the memory manager's tracking vector. The vector stores only pointers to objects, not the objects themselves. Therefore, resizing or compacting this vector does not move live TinyClojure objects; it only changes the collector's bookkeeping data.

The collector is therefore non-moving with respect to TinyClojure objects: live objects keep the same memory address for their entire lifetime. This simplifies the runtime, because generated code and runtime objects can safely store ordinary `Object*` references.

### 3.6.6 Limitations

The collector is intentionally simple. It is sufficient for reclaiming objects that become unreachable between top-level evaluations, but it is not a fully precise collector for arbitrary execution points.

The main limitation follows from the collection-point policy described in Section 3.6.3. Temporary values in generated code are not tracked precisely, so allocation-heavy code may retain objects until the current top-level evaluation finishes.

The collector is also non-generational and non-compacting. It does not attempt to group objects by age, reduce fragmentation, or optimize for short-lived allocations. More precise root tracking, stack maps, generational collection, or integration with LLVM's garbage-collection support are possible directions for future work.

### 3.6.7 Alternative garbage-collection approaches

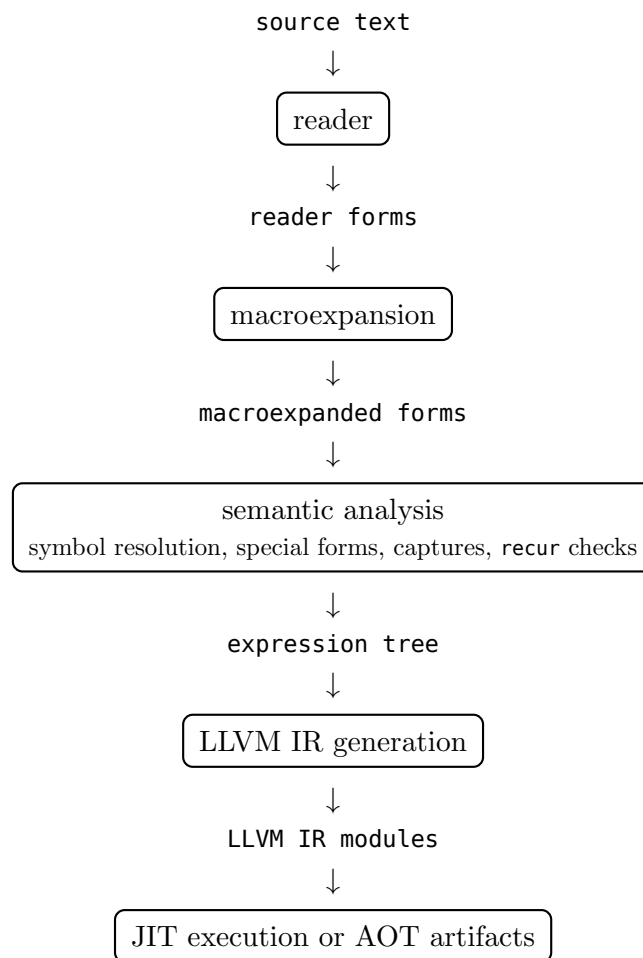
Instead of implementing a small custom collector, TinyClojure could use an existing garbage-collection framework. One option is the Boehm-Demers-Weiser garbage collector, also known as BDWGC. It is a conservative collector for C and C++ programs and can be used as a garbage-collecting replacement for `malloc` or C++ allocation [22]. This would reduce implementation effort, because the runtime would not need to implement its own tracing and sweeping logic. However, it would also make TinyClojure depend on conservative scanning rather than on the explicitly described object graph and root model used in this thesis.

Another possible option is MMTk, a memory-management toolkit for language runtimes. MMTk provides reusable garbage-collection infrastructure for language implementers and supports multiple runtime bindings [23]. Using MMTk would be closer to building a more serious language runtime, but it would require designing and maintaining an integration layer between TinyClojure and MMTk. This is beyond the scope of this thesis.

For these reasons, TinyClojure uses a small custom mark-and-sweep collector. This keeps the memory-management design simple enough to describe and implement within the thesis, while still making the required runtime responsibilities explicit.

This differs from Jank, which currently uses BDWGC [24] and has discussed MMTk as a possible future direction [25].

# Compiler implementation



■ **Figure 2** Overview of the TinyClojure compiler pipeline. Boxes represent compiler phases, while labels between them represent intermediate program representations. The generated LLVM IR modules are either added to the running JIT session or written as AOT artifacts.

This chapter describes the implementation of the TinyClojure compiler. It starts with the reader and macroexpansion, then describes semantic analysis, function compilation, and LLVM IR generation.

The overall pipeline is shown in Figure 2.

The rest of this chapter follows this pipeline. Section 4.1 describes the reader, which turns source text into TinyClojure forms. Section 4.2 describes how macros rewrite these forms before analysis. Section 4.3 explains how macroexpanded forms are converted into compiler-specific expression objects. Section 4.4 describes the compilation of function objects and closures. Finally, Section 4.5 describes the generation of LLVM IR modules.

The generated IR does not implement the whole language by itself. It relies on the runtime system for dynamic values, Vars, function invocation, list operations, memory allocation, and module loading.

## 4.1 Reader

The reader is the first stage of the pipeline shown in Figure 2. Its input is source text and its output is a sequence of TinyClojure forms represented as runtime values.

The implementation processes the input character by character. White-space and comments are skipped, delimiters introduce list forms, and other token sequences are parsed as literals or symbols. Errors such as unterminated strings, invalid character literals, invalid escape sequences, unmatched delimiters, or numeric literals outside the supported range are reported at this stage.

The reader is also responsible for reader-level shorthand forms. Quote, syntax quote, unquote, unquote-splicing, and auto-gensym syntax are recognized during reading. These constructs are translated into ordinary TinyClojure forms or runtime data structures before macroexpansion begins.

### 4.1.1 Syntax quote lowering

Syntax quote is implemented by the reader as a source-level transformation. The reader does not evaluate the syntax-quoted expression. Instead, it rewrites it into ordinary TinyClojure forms that construct the resulting data structure when evaluated later.

A syntax-quoted list is lowered recursively. Nested lists are lowered using the same rules, while non-list values are quoted. The result is expressed as a call to `concat`, which appends lists from left to right. Each argument of `concat` must evaluate to a list or `nil`.

Each element of the original syntax-quoted list is lowered as follows:

A syntax-quoted list is lowered to a call to `concat`. The helper function `concat` constructs a list by appending the elements of its arguments from left to right. Each argument must be a list or `nil`; `nil` contributes no elements.

- An ordinary form is lowered to a one-element list containing the quoted form.
- An unquoted form, written `~form`, is lowered to a one-element list containing the unquoted form itself.
- An unquote-spliced element, written `~@form`, is lowered to the unquoted form itself, because its result should be spliced into the surrounding list.

```
`(+ ~a 1)
```

is lowered to:

```
(concat
  (list (quote +))
  (list a)
  (list (quote 1)))
```

Here, `a` is not quoted, because it appears inside an unquote form.

Similarly:

```
`(+ ~@args 4)
```

is lowered to:

```
(concat
  (list (quote +))
  args
  (list (quote 4)))
```

Here, `args` is passed directly to `concat`, because `~@args` splices its elements into the surrounding list.

## 4.2 Macroexpansion

Macroexpansion is performed after reading and before semantic analysis. It rewrites source forms by invoking macro functions and using their return values as replacement forms. The resulting form is expanded again, so macroexpansion is recursive. This allows macros to expand into forms that themselves contain macro calls. After macroexpansion finishes, the analyzer processes the final expanded form as ordinary TinyClojure code.

Macroexpansion is important because several convenient language constructs are implemented as macros rather than as primitive special forms. For example, core-library forms such as `fn`, `defn`, `let`, `loop`, `when`, `cond`, `and`, and `or` are expanded into smaller primitive forms such as `fn*`, `def`, `let*`, `loop*`, `if`, and `do`.

The implementation therefore keeps the compiler kernel smaller. The analyzer only needs to understand the primitive special forms, while higher-level constructs can be defined in the core library.

### 4.3 Semantic analysis

Semantic analysis is the stage that turns macroexpanded forms into compiler-specific expression objects. The reader and macroexpander operate on generic TinyClojure values such as symbols, lists, and literals. These values preserve the structure of the source program, but they do not yet say whether a list represents a special form, a macro call, or an ordinary function call.

Semantic analysis assigns this meaning. It recognizes special forms, resolves local and global names, records captured variables, validates `recur` usage, and rejects forms outside the implemented language subset. The result is an expression tree used by later compiler phases, not LLVM IR.

The main analysis steps are summarized in Table 2.

| Step                          | Purpose                                                                                                            | Result                               |
|-------------------------------|--------------------------------------------------------------------------------------------------------------------|--------------------------------------|
| Special-form recognition      | Determine whether a list represents a special form, macro-expanded form, or ordinary function call.                | Expression-specific analysis path    |
| Symbol resolution             | Resolve symbols as local bindings or global Vars.                                                                  | Local references and Var references  |
| Lexical-scope tracking        | Maintain the set of currently visible local bindings while analyzing nested forms.                                 | Scoped local environment             |
| Capture analysis              | Detect references from an inner function to bindings defined in an outer function.                                 | Captured-variable list for functions |
| Function analysis             | Validate overloads, parameter lists, fixed arities, and variadic arities.                                          | Analyzed function expression         |
| <code>recur</code> validation | Check that <code>recur</code> appears only in a valid recursion position and with the correct number of arguments. | Validated recursion target           |

**Table 2** Main semantic-analysis steps performed by TinyClojure.

For example, after macroexpansion, a form such as

```
(let* (x 1)
      (+ x 2))
```

is no longer treated as an arbitrary list. Semantic analysis recognizes `let*` as a special form, introduces a local binding for `x`, analyzes the initializer `1`, and then analyzes the body in an environment where `x` is visible. The body form `(+ x 2)` is analyzed as a function call: the symbol `+` becomes a global Var reference or a candidate for direct linking, the symbol `x` becomes a local binding reference, and `2` becomes an integer literal expression.

### 4.3.1 Symbol resolution

When analyzing a symbol, the analyzer first searches the current lexical environment. If the symbol names a local binding, the analyzer produces an expression that reads that local value. If no local binding exists, the symbol is treated as a global Var reference.

This distinction is important for code generation. Local bindings can be accessed directly through the current function's environment, while global symbols require interaction with the runtime Var system. In some cases, a global Var reference may also be used for direct linking, as described later in this chapter.

For example, in the body of

```
(let* (x 1)
      (+ x 2))
```

the symbol `x` resolves to the local binding introduced by `let*`, while `+` resolves to a global Var or to a directly linkable function target. This distinction is preserved in the analyzed expression tree and later affects code generation.

### 4.3.2 Lexical scopes and captured variables

During semantic analysis, the analyzer maintains a lexical environment describing currently visible local bindings. Each function, `let*`, and `loop*` form extends this environment with new bindings. When a symbol is analyzed, the analyzer first searches this lexical environment before falling back to the global Var environment.

If a function body refers to a local binding defined outside that function, the binding cannot be accessed as an ordinary local variable. The analyzer therefore records it as a captured variable. This information is stored in the analyzed function expression and later used during code generation to construct the closure environment.

Capture analysis is performed before LLVM IR generation. This keeps code generation simpler: by the time a function is emitted, the compiler already

knows which values must be stored in the closure object and which captured-value index corresponds to each referenced symbol.

For example:

```
(fn* [x]
  (fn* [y]
    (+ x y)))
```

When analyzing the inner function, the symbol `y` resolves to a parameter of the inner function, while `x` resolves to a binding from the surrounding function. The analyzer therefore records `x` as a captured variable of the inner function. Code generation later uses this information to store `x` in the closure environment.

### 4.3.3 Special forms

Lists are analyzed according to their first element. If the first element names a special form, the analyzer checks the structure of the form and creates the corresponding expression object. For example, `if` forms are checked for the correct number of branches, `let*` and `loop*` forms introduce local bindings, and `fn*` forms introduce function objects.

This phase is also where malformed special forms are rejected. Examples include an invalid binding list, an incorrect number of arguments to `if`, duplicate function arities, or a `recur` form used outside a valid recursion target.

### 4.3.4 Function analysis and closures

Function analysis introduces a new lexical environment for the function parameters and body. Each overload is analyzed separately, because each overload has its own parameter list, body, and recursion target.

Fixed arities and variadic arities are checked according to the rules described in the language specification. The analyzer rejects duplicate fixed arities and invalid combinations of fixed and variadic overloads. It also records information needed by function compilation, such as the number of required parameters, whether an overload is variadic, and which captured variables are used by the function.

When the analyzer encounters a `fn*` form, it analyzes the function body and constructs a function expression for it. In `TinyClojure`, this step also triggers compilation of the function expression into a separate LLVM module. The generated module is added to the active JIT session. If AOT compilation is requested, the same module is also emitted as an AOT artifact.

This is an implementation choice rather than a property of the source language. It reflects the module structure used by `TinyClojure`: function bodies

are compiled as separate modules, while the surrounding expression only needs a runtime function value referring to the compiled function. As a result, nested function forms are compiled eagerly when they are encountered during semantic analysis. Later code generation for the outer form can then emit a reference to the already compiled function module rather than compiling the nested function body at the call site.

### 4.3.5 **Recur analysis**

The analyzer tracks the current recursion target while analyzing `loop*` and function bodies. A `recur` form is valid only when such a target exists and when the number of supplied values matches the target's expected arity.

This check is performed during semantic analysis rather than during LLVM IR generation, because invalid `recur` forms are language errors. Once validated, `recur` can later be lowered to a branch to the beginning of the corresponding loop or function overload, rather than to an ordinary function call.

## 4.4 **Function compilation**

Functions are one of the central constructs of TinyClojure. A function is both a piece of executable code and a runtime value that can be stored in a `Var`, passed as an argument, returned from another function, or captured by a closure. The compiler must therefore generate executable LLVM IR and also provide the runtime-level information needed to construct a function object.

A `fn*` form is analyzed into a function expression containing one or more overloads. Each overload is the body associated with one supported arity case. The compiler generates code for the individual overload bodies, a runtime-callable dispatch entry compatible with the calling convention described in Section 3.3, and a method-table global used by the runtime function object.

The dispatch entry performs arity dispatch. For fixed-arity overloads, it checks whether the supplied argument count matches the overload exactly. For a variadic overload, it checks whether the number of supplied arguments is at least the number of required parameters. If a matching overload is found, control is transferred to the corresponding overload body. If no overload matches, the function reports an arity error through the runtime.

For the purposes of this description, a generated function module can be viewed as containing three main parts:

| Part                | Purpose                                                                                                                                               |
|---------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------|
| Method-table global | Stores the runtime operations for the function object, including the call entry used by invocation.                                                   |
| Dispatch entry      | Receives <code>self</code> , <code>argc</code> , and <code>argv</code> according to the uniform calling convention and selects the matching overload. |
| Overload bodies     | Contain the compiled code for individual fixed-arity or variadic overloads. Parameters and captured values are made available as local bindings.      |

■ **Table 3** Main parts of a generated function module.

The exact LLVM IR is described later in Section 4.5. The important point here is the separation of responsibilities: the function module contains the executable code, dispatch entry, and method-table global for a function. The surrounding expression or file-level module can then create a runtime function object that refers to this compiled function module and use it as an ordinary TinyClojure value.

#### 4.4.1 Function modules

TinyClojure compiles each function into its own LLVM module. This keeps function code separated from file-level initialization code and allows functions to be compiled, loaded, and referenced independently. A source file module is then responsible for creating function objects and binding them to Vars or storing them in other runtime values.

This design is similar in purpose to the separation between function classes and namespace initialization code in JVM Clojure. The function module contains the executable code for the function, while the file-level module performs top-level actions such as declaring Vars, creating function objects, binding roots, and executing ordinary top-level expressions.

Each generated function module contains the code needed for the function’s supported arities. It also exposes a runtime-callable entry point compatible with the uniform calling convention described in Section 3.3.

#### 4.4.2 Arity dispatch

For multi-arity functions, the compiler emits a single runtime-callable entry point and separate code paths for individual overloads. The entry point imple-

ments the arity-selection rules defined in Section 2.5.5 and transfers control to the matching overload body.

Variadic overloads additionally require constructing the rest argument. The required parameters are read directly from the argument array, while the remaining arguments are collected into a TinyClojure list and bound to the rest parameter.

If no overload matches, the entry point calls a runtime helper that reports an arity error.

### 4.4.3 Parameters and local bindings

Inside an overload body, parameters are represented as local bindings. Fixed parameters are initialized from the argument array received through the runtime calling convention. The body expressions are then emitted in order, and the value of the last expression becomes the return value of the overload.

This means that the internal representation of a function body is similar to a `let*` form whose bindings are initialized from call arguments. The difference is that the binding values are supplied by the function invocation mechanism rather than by explicit initializer expressions in the source program.

### 4.4.4 Captured values

If a function refers to local bindings from an outer lexical scope, the analyzer records these bindings as captures. During code generation, the compiler creates a function object that stores the captured values in its runtime payload.

The captured environment is represented as an array of `Object` pointers. Each captured binding is assigned an index in this array during semantic analysis. When generated code needs to read a captured value, it loads the corresponding element from the function's captured environment.

The runtime calling convention passes the function object itself as the first argument. This makes the closure object available when the function is invoked. For overloads that use captured values, the dispatch function extracts the captured environment from the function object and passes it to the compiled overload body. Overloads that do not use captures do not need this extra argument.

This allows captured values to remain available even after the scope that created the closure has returned. From the point of view of the generated overload body, captured values behave similarly to additional read-only local bindings initialized from the closure object.

## 4.5 LLVM IR generation

LLVM IR generation is the final lowering stage in Figure 2. It translates the analyzed expression tree into LLVM modules that call into the runtime system.

After semantic analysis and function compilation setup, the compiler emits LLVM IR from the analyzed expression tree. Each expression node appends LLVM instructions to the current function and returns the LLVM value representing the result of the expression.

Because TinyClojure is dynamically typed, most expression results are represented as pointers to runtime objects. Generated LLVM IR therefore frequently calls runtime helper functions for object construction, dynamic checks, Var operations, function invocation, and error reporting.

### 4.5.1 Emission model

After semantic analysis, the compiler no longer works with raw reader forms. Instead, it emits LLVM IR from analyzed expression objects. Each expression object represents one kind of TinyClojure expression, such as a literal, local binding reference, Var reference, function call, conditional, or a local binding form.

The code generator maintains a context that wraps the LLVM API objects and compiler state needed during emission. Expression objects use this context to append instructions, access local bindings, and call runtime helper functions.

Most expression emission functions return an LLVM value representing the result of the TinyClojure expression. Since TinyClojure is dynamically typed, this result is usually a pointer to a runtime `Object`. This uniform result type makes it possible to compile nested expressions without knowing their exact runtime types statically.

Some expressions are emitted for their side effects. For example, top-level definitions update Vars in the runtime global environment. Even in these cases, the expression still produces a TinyClojure value according to the language rules.

There are also expressions whose emitted control flow does not continue normally. In the current implementation, `recur` is the main example: it transfers control to the nearest recursion point and therefore does not produce a value for the following instruction. The code generator must handle such cases explicitly, because there is no ordinary result value that can be used by an enclosing expression.

A similar situation arises for runtime-error paths. TinyClojure does not provide user-level exception handling; compiler-emitted runtime-error paths

call the external runtime function `tinyclj_rt_error` and then emit an LLVM `unreachable` instruction.

### 4.5.2 Expression evaluation in the REPL

Most of this section describes LLVM IR generation for compiled code. The interactive REPL also needs to evaluate individual forms entered by the user. For this purpose, analyzed expressions provide an `Expr::eval` operation.

This operation is not the main execution mechanism for AOT-compiled modules. It is used by the runtime when evaluating forms interactively. The runtime first macroexpands and analyzes the input form. Some forms can then be handled directly from the analyzed expression tree, while ordinary expressions are compiled by wrapping them in a zero-argument function.

Conceptually, REPL evaluation follows this structure:

```
form = macroexpand(form)

if form is a do block:
    evaluate subforms sequentially
else if form can be evaluated directly:
    analyze(form).eval()
else:
    wrapper = (fn* () form)
    fn_object = analyze(wrapper).eval()
    fn_object.invoke()
```

■ **Listing 12** Schematic evaluation strategy used by the TinyClojure REPL.

The wrapper function is an implementation technique. It allows the runtime to reuse the ordinary function-compilation pipeline for expressions entered in the REPL. For example, a user-entered call such as `(+ 1 2)` can be wrapped as a zero-argument function, compiled, and invoked. Direct expression evaluation remains useful for forms whose effect is managed by the runtime or global environment, such as top-level definitions.

In the current implementation, top-level `def` forms are evaluated directly, and `do` forms are handled by evaluating their subforms sequentially. Other forms can be wrapped in a generated zero-argument function so that their execution uses the same LLVM-based path as compiled code.

### 4.5.3 Literals and runtime values

Literal expressions are lowered to runtime values. Since most TinyClojure values are represented as pointers to runtime `Objects`, literal emission usually produces an `Object*`.

The value `nil` is the simplest case: it is emitted as a null object pointer. Boolean literals are emitted using runtime singleton objects. The runtime initializes global objects representing `true` and `false`, and generated code obtains the corresponding object pointer when a boolean literal is needed.

Numeric literals are emitted by calling runtime construction helpers, described in Section 3.1.2. Integer literals are passed as 64-bit integer values to `tc_integer_new`, and floating-point literals are passed as 64-bit floating-point values to `tc_double_new`. The helper then returns the corresponding boxed runtime object. For integers, this also allows the runtime to reuse cached integer objects when the literal value is within the configured cache range.

For example, numeric literals can be emitted schematically as shown in Listing 13.

```
%integer_value = call ptr @tc_integer_new(i64 42)
%double_value  = call ptr @tc_double_new(double 3.14)
```

■ **Listing 13** Schematic LLVM IR for constructing numeric literals.

String and symbol literals require textual data to be available to the generated code. The compiler emits the textual payload as a private LLVM string constant and passes a pointer to this constant to a runtime helper function. The runtime helper constructs the corresponding TinyClojure string or symbol object.

```
@str = private unnamed_addr constant [6 x i8] c"hello\00"
@sym = private unnamed_addr constant [4 x i8] c"foo\00"
...
%string_value = call ptr @tc_string_new(ptr @str)
%symbol_value = call ptr @tc_symbol_new(ptr @sym)
```

■ **Listing 14** Schematic LLVM IR for constructing string and symbol literals.

This design keeps the generated LLVM IR independent of the internal layout of runtime payloads. For example, the compiler does not need to know how a `TCString` or `TCSymbol` stores its data internally. It only needs to provide the literal payload and call the appropriate runtime helper. The helper allocates or retrieves the runtime object and returns it through the uniform `Object*` representation.

#### 4.5.4 Local bindings

For local bindings, the code generator maintains a table mapping analyzed local symbols to LLVM values. When emitting a `let*` form, the initializer

expression is emitted first. The resulting LLVM value is then registered in the current code-generation context before the body is emitted. After the body has been emitted, the previous binding state is restored.

Conceptually, code generation for a single `let*` binding follows this structure:

```
auto value = emit(initializer);
auto old_binding = context.find(symbol);

context[symbol] = value;
auto result = emitBody(body);

// Restore the previous lexical scope.
if (old_binding.has_value()) {
    context[symbol] = old_binding.value();
} else {
    context.erase(symbol);
}

return result;
```

This pseudocode illustrates the scope-management idea. The actual implementation also has to handle multiple bindings, nested scopes, and shadowing. If an inner binding uses the same symbol name as an outer binding, the inner binding temporarily hides the outer one. When the inner scope ends, the previous binding becomes visible again.

For example, when generating code for the TinyClojure form

```
(let* (x 10)
  (+ x x))
```

the initializer `10` is emitted first. As described in Section 3.1.2, numeric literals are lowered by calling runtime construction helpers. Conceptually, this produces a call to a helper such as `tc_integer_new`, which returns an `Object*` representing the integer value.

The code generator then records that the local symbol `x` is associated with this LLVM value. When `x` is referenced in the body, the code generator looks it up in the local binding table and reuses the same value. No runtime name lookup is performed for `x`.

Schematically, if `+` is compiled as a direct call to an addition helper, the generated LLVM IR may have the following shape:

```
%x = call ptr @tc_integer_new(i64 10)
%result = call ptr @tc_builtin_add(ptr %x, ptr %x)
```

The exact emitted call sequence depends on whether `+` is invoked dynamically or through direct linking. The important point is that both references to `x` reuse the same LLVM value associated with the local binding.

For multiple bindings, initializers are emitted from left to right, and each binding is made visible to the following initializers and to the body.

### 4.5.5 Vars

Global Vars are emitted differently from local bindings. A local binding is resolved entirely by the compiler and is represented by an LLVM value in the current code-generation context. A global symbol, however, refers to a runtime Var object. This preserves the dynamic behavior of TinyClojure Vars: the Var object remains stable, while its root value may change during module loading or interactive evaluation.

For each global Var referenced by a compiled module, the compiler creates an LLVM global variable that stores a pointer to the corresponding runtime Var object. This LLVM global does not store the Var's root value directly. It only stores the address of the runtime Var object. The root value remains inside the runtime Var and is accessed through runtime helper functions.

For example, a top-level definition such as `(def x 10)` may be represented schematically by an LLVM global slot:

```
@var_x = internal global ptr null
```

During module initialization, generated code declares or retrieves the runtime Var object and stores it into this slot:

```
define void @my_module__init() {  
  entry:  
    %var = call ptr @tc_runtime_declare_var(ptr @symbol_x)  
    store ptr %var, ptr @var_x  
  
    %value = call ptr @tc_integer_new(i64 10)  
    call void @tc_var_bind_root(ptr %var, ptr %value)  
  
    ret void  
}
```

Later references to the global symbol load the Var object from the LLVM global slot and dereference its root value:

```
%var = load ptr, ptr @var_x  
%value = call ptr @tc_var_get_root(ptr %var)
```

This is different from local bindings. A local binding such as `x` in a `let*` form is not looked up at runtime; it is mapped directly to an LLVM value by the code generator. A global symbol must go through the Var object unless the compiler applies an optimization such as direct linking for a known function call.

The global slot is initialized by module initialization code rather than at LLVM module creation time, because the Var object is managed by the

TinyClojure runtime. The runtime must either create the Var or return the existing Var if it has already been declared.

### 4.5.6 Function calls

During semantic analysis, a non-empty list that is not a special form or macro invocation is analyzed as a function call. The expression in function position is analyzed separately from the argument expressions. This produces an invocation expression containing the analyzed target expression and the analyzed arguments. During code generation, this invocation expression is emitted either using the dynamic call path or, when possible, using direct linking.

Function calls are emitted by first evaluating the call target and then evaluating the argument expressions from left to right. The general call path is dynamic: the compiler emits code that treats the call target as an arbitrary runtime `Object` and checks at runtime whether it is callable.

The order of emission is important. The call target expression is emitted first, followed by all argument expressions. Only after the arguments have been evaluated does the generated code check whether the target object is `nil`. This preserves the evaluation order of the language: argument expressions are evaluated even if the call target later turns out not to be callable.

After the arguments have been emitted, the generated code checks that the target object is not `nil`. It then loads the object's method table and retrieves the call function pointer from it. If the call function pointer is null, the object is not callable and a runtime error is reported. Otherwise, the generated code builds the argument array expected by the runtime calling convention and calls the retrieved function pointer.

Conceptually, dynamic invocation follows this sequence:

```
target = emit call target
args = emit argument expressions from left to right

if target == nil:
    report runtime error

method_table = target->m_MethodTable
call_fn = method_table->m_Call

if call_fn == nil:
    report runtime error

argv = build argument array from args
result = call_fn(target, argc, argv)
```

This dispatch path allows any runtime object to decide whether it is callable through its method table. Function objects provide a call function, while ordinary values such as numbers, strings, or lists do not. The dynamic path is therefore necessary for higher-order functions, closures, and calls through local variables.

#### 4.5.7 Direct linking

Direct linking was introduced in Section 1.5.2 as Clojure’s term for compiling selected function calls directly to known targets. Runtime-level motivation was described in Section 3.3.5. This section describes how TinyClojure emits this optimization in LLVM IR.

In TinyClojure, each compiled function module defines a static method table for the generated function object. The method table contains, among other operations, the function’s call entry point. In the generated LLVM IR, these method-table globals use names ending in `__vtable`. In this section, *vtable* refers to this generated method-table symbol; it is not a C++ virtual table.

For a function module named `foo`, the generated method table may be represented schematically as shown in Listing 15.

```
@foo__vtable = global %MethodTable {  
  ptr @foo__call,  
  ...  
}
```

■ **Listing 15** Schematic LLVM IR for a generated function method table.

A dynamically emitted function call evaluates the callee expression, obtains the resulting runtime function object, loads the method table from that object, and then loads the call entry from the method table. Direct linking skips the runtime Var dereference and the method-table load from the function object. Instead, semantic analysis has already selected the target function module, so code generation can refer directly to the known method-table global.

If the method table is defined in another LLVM module, the current module declares it as an external global. A direct-linked call then has the schematic shape shown in Listing 16.

```
@foo__vtable = external global %MethodTable
...
%call_slot = getelementptr %MethodTable, ptr @foo__vtable,
                    i32 0, i32 CALL_FIELD
%call_fn = load ptr, ptr %call_slot

; arguments are still evaluated normally
%result = call ptr %call_fn(ptr %self, i64 %argc, ptr %argv)
```

■ **Listing 16** Schematic LLVM IR for a direct-linked function call.

Here, `CALL_FIELD` denotes the index of the call entry inside the method-table structure. The exact numeric index depends on the LLVM representation of `MethodTable`.

Compared to dynamic invocation, this avoids loading the method table from the runtime target object. The call entry point is obtained from the known LLVM method-table global instead.

Direct linking also changes how later Var redefinitions are observed. A dynamically emitted call obtains the current root value of the Var at runtime. A direct-linked call bypasses that runtime dereference and continues to use the function selected during compilation. For this reason, direct linking is an optimization for calls whose target is intended to be fixed for the compiled module.

### 4.5.8 Built-in functions

Built-in functions use the same callable-object interface as compiled TinyClosure functions, but their method tables are created by the C++ runtime rather than by generated LLVM function modules. In the implementation, the runtime uses a C++ template parameterized by the native call function pointer to create a static method table for each built-in function.

The following simplified code shows the registration pattern:

```

template<CallFn Fn>
struct BuiltinFunctionVTable {
    static constexpr MethodTable value = {
        .m_CallFn = Fn,
        .m_EqualsFn = nullptr,
        .m_ToStringFn = nullptr,
        .m_ToEdnFn = nullptr
    };
};

template<CallFn Fn>
void Runtime::defn(const std::string &name, const char
*nativeFnName) {
    Runtime &rt = Runtime::getInstance();
    auto var = rt.declareVar(name);
    tc_var_bind_root(var, tc_function_new(
        &BuiltinFunctionVTable<Fn>::value,
        nativeFnName));
}

#define TC_DEFN(name, nativeFn) defn<nativeFn>(name, #nativeFn)

void Runtime::init() {
    TC_DEFN("builtin_binary_add", tinyclj_rt_add);
    TC_DEFN("builtin_binary_sub", tinyclj_rt_sub);
    ...
}

```

The macro `TC_DEFN("builtin_binary_add", tinyclj_rt_add)` registers a Var named `builtin_binary_add` and binds its root value to a function object. The function object uses the static method table instantiated for `tinyclj_rt_add`, and it also stores the native symbol name `"tinyclj_rt_add"`.

All built-in native functions are declared with C linkage, which makes their symbol names easier to reference from generated LLVM IR. Therefore, built-ins can participate in the same direct-linking mechanism as compiled functions. When the compiler resolves a call to a known built-in function, it can use the stored native symbol name and emit a direct call path instead of going through dynamic method-table dispatch.

For compiled TinyClojure functions, direct linking uses the generated function module's method-table symbol. The symbol name is derived from the function module name by appending the suffix `"__vtable"`. Built-in functions are handled differently. They are registered by the runtime as function objects whose stored native symbol name refers to an external C-linkage runtime function, such as `tinyclj_rt_add`. When the compiler direct-links such a built-

in, it can use this stored native symbol name directly instead of referring to a generated method-table symbol.

In both cases, the surrounding calling convention remains the same: the call receives the target function object, the argument count, and the argument array. The difference is only how the call entry point is obtained.

#### 4.5.9 **do blocks**

A **do** block evaluates several expressions in sequence and returns the value of the last expression. Code generation for **do** therefore emits each expression in order. The results of all expressions except the last one are ignored, although their emitted instructions and side effects remain part of the generated code.

Conceptually, a form such as

```
(do
  expr-1
  expr-2
  expr-3)
```

is emitted as:

```
emit expr-1 and discard its result
emit expr-2 and discard its result
result = emit expr-3
return result
```

This pattern is used not only for explicit **do** forms. Other constructs whose bodies contain multiple expressions are emitted in the same way. For example, the body of a **let\***, **loop\***, or function overload is treated as a sequence of expressions whose result is the value of the last expression. The semantic analyzer ensures that expressions that terminate control flow, such as **recur**, appear only in positions where this sequencing rule is valid.

#### 4.5.10 **Conditionals**

An **if** expression is lowered to explicit LLVM control flow. The condition expression is emitted first. Since TinyClojure follows Clojure-like truthiness rules, only **nil** and **false** are treated as falsey values. The generated code therefore compares the condition result with the null object pointer and with the runtime singleton representing **false**.

Based on this check, the compiler emits a conditional branch to either the then block or the else block. Both branches are emitted as ordinary expressions. Since **if** itself is also an expression, the values produced by the two branches must be joined after the branch.

This join is represented using an LLVM phi node in the merge block. The phi node selects the value produced by the branch that was actually executed.

Conceptually, the form

```
(if test
  then-expr
  else-expr)
```

is lowered to the following control-flow shape:

```
test-value = emit test

if test-value is neither nil nor false:
  branch then-block
else:
  branch else-block

then-block:
  then-value = emit then-expr
  branch merge-block

else-block:
  else-value = emit else-expr
  branch merge-block

merge-block:
  result = phi [then-value, then-block], [else-value, else-block]
```

If the source form does not contain an else branch, the compiler emits nil as the else value.

The compiler must also account for branches that do not continue normally. For example, if one branch ends with `recur` or a runtime-error path, that branch does not produce a value for the merge block. In such cases, the phi node only receives incoming values from branches that actually reach the merge block. If at least one branch reaches the merge block, the `if` expression emission returns the phi node, which can be treated as a value by the surrounding code generator.

#### 4.5.11 Loops and recur

The `loop*` form is compiled as an explicit local recursion point. During semantic analysis, the analyzer records that the body of the `loop*` form may contain `recur` expressions targeting this loop. Code generation then lowers this structure to LLVM basic blocks and phi nodes.

A `loop*` form first evaluates its initializer expressions from left to right. The resulting values become the initial values of the loop bindings. The compiler then emits a loop body block. At the beginning of this block, each loop binding

is represented by an LLVM phi node. The phi node selects either the initial value, when control enters the loop for the first time, or the value supplied by a later `recur`.

The loop body is emitted using the same sequencing rule as a `do` block: expressions are emitted in order, and if the body exits normally, the value of its last expression becomes the value of the `loop*` form.

Conceptually, a form such as

```
(loop* (x x-init
      y y-init)
  body)
```

is lowered to the following control-flow shape:

```
entry:
  x-initial-value = emit x-init
  y-initial-value = emit y-init
  branch loop-body

loop-body:
  x = phi [x-initial-value, entry], [x-next-value, recur-block]
  y = phi [y-initial-value, entry], [y-next-value, recur-block]
  result = emit body
  if body continues normally:
    branch loop-exit

loop-exit:
  loop-result = result
```

A `recur` expression targeting the loop is emitted as a back edge to the loop body. It first evaluates its argument expressions from left to right. During code generation, these values are registered as incoming values of the corresponding phi nodes, and the generated code emits an unconditional branch back to the loop body.

Conceptually, a `recur` form such as `(recur y (+ x 1))` is lowered to the following control-flow shape:

```
recur-block:
  x-next-value = y
  y-next-value = emit invoke(+, [x, 1])
  branch loop-body
```

Because `recur` transfers control back to the recursion point, it does not produce an ordinary result value for the following instruction. The surrounding code generator must therefore treat it as a terminating expression.

For example:

```
(loop* (x 3)
  (if (= x 0)
    x
    (recur (- x 1))))
```

The initializer 3 provides the first incoming value of the phi node for `x`. The `recur` expression computes `(- x 1)` and provides the second incoming value, coming from the block that branches back to the loop body. The condition reads the current value selected by the phi node. If the condition is true, the loop exits with that value; otherwise, control reaches the `recur` back edge.

As a small simplification, semantic analysis detects `loop*` forms whose body contains no `recur` targeting that loop. Such a form does not need a recursion point, back edge, or phi nodes. The analyzer can therefore lower it to the same representation as `let*`, because without `recur` the loop bindings are initialized once and the body is evaluated once. This avoids generating unnecessary loop control flow for a form that cannot actually loop.

#### 4.5.12 Runtime helper calls and error paths

Generated LLVM IR does not implement all runtime behavior directly. Operations that depend on TinyClojure runtime objects are emitted as calls to external runtime helper functions. These helpers allocate boxed values, manipulate Vars, construct lists, perform arithmetic and comparison operations, invoke callable values, and report runtime errors.

This keeps the generated IR relatively small and separates language-level runtime behavior from low-level code generation. For example, the compiler does not emit the internal logic for integer allocation, Var dereferencing, or list construction directly. Instead, it emits calls to runtime functions with the appropriate `Object*` arguments.

There is a distinction between ordinary user-written calls and compiler-emitted error paths. A program may call an error-reporting function explicitly, in which case the call is emitted like any other function call and its result is treated as an ordinary expression result. In contrast, when the compiler emits a check for an invalid runtime operation, such as invoking `nil`, invoking a non-callable object, or failing an arity check, it emits a dedicated error block. This block calls the runtime error helper using the ordinary TinyClojure calling convention and then ends with LLVM `unreachable`.

A simplified arity-check error path is shown in Listing 17.

```

; argc contains the number of supplied arguments
switch i32 %argc, label %argcnt_default [
  i32 0, label %argcnt_0
  i32 1, label %argcnt_1
]

argcnt_default:
  %is_enough_args_for_variadic_case = icmp uge i32 %argc, 3
  br i1 %is_enough_args_for_variadic_case,
      label %variadic_case,
      label %argcnt_wrong

argcnt_wrong:
  %argv_error = alloca ptr, i64 1
  %message = call ptr @tc_string_new(ptr @wrong_arity_message)
  store ptr %message, ptr %argv_error
  call ptr @tinyclj_rt_error(ptr null, i64 1, ptr %argv_error)
  unreachable

```

■ **Listing 17** Schematic LLVM IR for a compiler-emitted arity error path.

Listing 17 shows a simplified arity-check error path. If none of the supported arity cases matches, control reaches the error block. The block calls `tinyclj_rt_error` using the ordinary TinyClojure calling convention. Since the helper expects its message as a TinyClojure value, the generated code first constructs a string object and passes it as the helper’s single argument. The following `unreachable` instruction records that execution does not continue normally after the error.

The current implementation does not maintain a full TinyClojure-level stack trace for runtime errors. Error paths report the detected error message and terminate execution, but they do not reconstruct the sequence of TinyClojure function calls that led to the error. Native debugging tools can still show the underlying C++ or generated-code frames, but this information is not presented as a source-level TinyClojure backtrace.

Providing a source-level backtrace would require additional runtime metadata, such as source locations, function names, and call-frame information for generated code. TinyClojure currently does not maintain this metadata, so runtime errors report the immediate error but not a full language-level call stack.

## Chapter 5

# Evaluation

This chapter evaluates TinyClojure from three perspectives. First, it describes the tests used to check that the implemented language subset behaves as expected. Second, it evaluates runtime performance using a set of small benchmark programs. Finally, it discusses the expressiveness and limitations of the implemented subset.

The evaluation is not intended to prove full compatibility with Clojure. TinyClojure deliberately implements only a restricted subset of the language. Instead, it focuses on the correctness of this subset, the performance trade-offs caused by the chosen runtime representation, and the kinds of programs that the subset can express.

### 5.1 Correctness testing

The implementation is tested using a combination of unit tests and integration tests. Unit tests cover individual compiler and runtime components. Integration tests execute complete TinyClojure programs by running the TinyClojure executable non-interactively: the test runner provides the input forms and compares the produced output with the expected output.

The same executable can also be used to start the interactive REPL, but the integration tests do not rely on manual interactive input.

#### 5.1.1 Reader unit tests

The reader is tested using a small custom C++ test executable. The tests do not use a third-party unit testing framework such as GoogleTest or Catch2. Instead, the executable initializes the TinyClojure runtime, calls the reader directly using `Runtime::readString`, and compares the returned runtime value with an expected value using the runtime equality function.

The reader test suite contains 43 test cases: 25 successful parsing cases and 18 expected-error cases. The successful cases cover integer and floating-point literals, booleans, `nil`, strings, symbols, character literals, and nested lists. Character-literal tests include ordinary characters, special named char-

acters such as `\newline`, octal character notation, and hexadecimal character notation.

The expected-error cases check that malformed input is rejected. These cases include missing character tokens, invalid octal and hexadecimal character literals, character values outside the supported byte range, unterminated strings, invalid string escape sequences, invalid numeric literals, unmatched parentheses, unexpected characters inside lists, and incomplete reader shorthand forms such as syntax quote, unquote, and unquote-splicing at the end of input.

These tests are useful because the reader is the first stage of the compiler pipeline. Errors in the reader would affect all later phases, so testing it separately makes failures easier to localize. The reader unit tests are complemented by the integration tests described below, which execute complete TinyClojure programs through the full compiler and runtime pipeline.

### 5.1.2 End-to-end comparison with JVM Clojure

End-to-end tests execute complete TinyClojure inputs and compare their observable output with JVM Clojure where the semantics overlap. The tests are driven by a Babashka [26] script, a lightweight Clojure scripting runtime. The script reads a file containing a sequence of Clojure forms, executes the same forms using JVM Clojure and TinyClojure, normalizes small syntax differences, and compares the resulting output line by line.

The main syntax difference handled by the script is that TinyClojure uses lists in places where Clojure normally uses vectors, such as parameter lists and local-binding forms. For example, the test input may contain the following Clojure form:

```
(let [x 1
      y 2]
  (+ x y))
```

This form can be executed directly by JVM Clojure. Before sending the input to TinyClojure, the script rewrites square brackets to parentheses:

```
(let (x 1
      y 2)
  (+ x y))
```

Both versions are expected to produce the same observable output:

```
3
```

The standard end-to-end comparison uses one input file containing 37 top-level forms. The file is executed once using JVM Clojure and once using TinyClojure, and the two complete outputs are compared line by line. In this

setup, the individual forms are not reported as separate test cases; the whole file is one integration test input.

The forms are written to be largely independent, so they could also be split into separate files in the future. Keeping them in one file makes the comparison script simple and keeps the output in a single place, but it also means that a failure can affect the comparison of later output lines. When such a failure happens, the script reports the first differing lines to make the cause easier to locate.

The compared output is not limited to the value of a single form. The input may contain multiple forms, and these forms may print to standard output while they are being evaluated. The resulting output therefore contains both printed output and values printed by the REPL-style evaluation loop. Comparing the whole output makes the test closer to how the implementation is used interactively.

TinyClojure uses lists for binding forms and parameter lists, while Clojure uses vectors in several of these positions. To make the same test file usable for both implementations, the script first sends the original file to JVM Clojure and then transforms square brackets to parentheses before sending the input to TinyClojure.

The script removes the initial JVM Clojure REPL banner from the Clojure output before comparison. If the outputs differ, the script reports the first differing lines, which makes integration failures easier to diagnose.

The end-to-end forms cover literals, conditionals, local bindings, shadowing, ordinary function calls, higher-order functions, closures, recursion, `loop*/recur`, top-level definitions, Var rebinding, multi-arity functions, and variadic functions. Some tests also exercise macros indirectly through the core library, because macroexpanded forms are compiled and executed as part of ordinary program evaluation.

### 5.1.3 Expected-failure tests

The test suite also contains expected-failure tests. These tests are separate TinyClojure input files that are expected to fail either during semantic analysis, LLVM IR generation, or runtime execution. Unlike the reader unit tests described above, these files are intended to be valid reader inputs. Their purpose is to test later stages of the compiler and runtime pipeline.

The current expected-failure suite contains 29 input files. The tested error categories are summarized in Table 4.

| Category                                         | Files | Examples                                                                                                                                         |
|--------------------------------------------------|-------|--------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>def</code> special form errors             | 3     | wrong number of arguments, definition name not being a symbol                                                                                    |
| <code>fn*</code> and function-form errors        | 6     | invalid parameter names, missing argument list, ambiguous arities, invalid variadic arities                                                      |
| <code>if</code> special form errors              | 2     | too few or too many arguments                                                                                                                    |
| <code>let*</code> binding errors                 | 3     | binding form not being a list, odd binding count, non-symbol binding name                                                                        |
| <code>loop*</code> and <code>recur</code> errors | 7     | invalid loop bindings, mismatched <code>recur</code> argument count, <code>recur</code> outside a loop, <code>recur</code> outside tail position |
| <code>var</code> special form errors             | 3     | wrong number of arguments, argument not being a symbol                                                                                           |
| Other compile-time and runtime errors            | 5     | calling <code>nil</code> , wrong function argument count, division by zero, incorrect runtime type                                               |

■ **Table 4** Expected-failure test categories.

The expected-failure tests do not require a specific non-zero process exit code. Instead, the test runner checks that executing the program produces an error report, either on standard error or through the implementation’s printed error output. This matches the current implementation, where errors are reported by printing diagnostic messages rather than by using a uniform exception or exit-status mechanism.

The purpose of these tests is therefore not to verify a precise operating-system-level failure convention. Rather, they check that invalid programs are detected and reported instead of being silently accepted or producing an ordinary result.

#### 5.1.4 AOT compilation and loading tests

Ahead-of-time compilation and module loading are tested separately from ordinary REPL-style evaluation. The test compiles a module in one TinyClojure run, then starts a second run and loads the compiled module.

The tested module contains a macro with an observable side effect during macroexpansion:

```
(defmacro side-effect-macro (form)
  (println "Macro called")
  form)

(def hello "hello")

(defn world ()
  (side-effect-macro "world"))

(println hello (world))
```

■ **Listing 18** AOT macroexpansion test module. The macro prints a message when it is expanded and then returns its argument unchanged.

Listing 18 is used to test when macroexpansion happens for AOT-compiled modules. The macro has an observable side effect: it prints "Macro called" during macroexpansion. The module also prints "hello world" during module initialization.

The test runner first executes:

```
(compile-module "example")
```

This compiles the module to persistent artifacts. During this run, macroexpansion happens, so the output must contain both "Macro called" and "hello world".

The test runner then starts a separate TinyClojure run and executes:

```
(load-module "example")
```

During this second run, the compiled module is loaded from the generated artifacts. The output must contain "hello world", because the module initialization code is executed, but it must not contain "Macro called". This shows that the macro was expanded during compilation, not during later loading of the compiled module.

This is a single targeted integration test for the AOT pipeline. It checks an important property of compiled modules: loading an AOT-compiled module executes the already generated initialization code, but does not repeat reader-level processing, macroexpansion, semantic analysis, or LLVM IR generation for the module source.

The test is not intended to be a complete test suite for all module-loading behavior. It focuses specifically on the interaction between AOT compilation, module initialization, and macroexpansion. Other module-loading scenarios, such as deeper dependency graphs, stale artifacts, and loading errors, are outside the scope of the automated tests used in this evaluation.

Overall, all implemented automated correctness tests passed at the time of evaluation. This includes the reader unit tests, the end-to-end comparison

with JVM Clojure, the expected-failure tests, and the targeted AOT loading test. These tests do not constitute a full conformance suite for Clojure, but they provide regression coverage for the implemented TinyClojure subset.

## 5.2 Performance

This section evaluates the runtime performance of TinyClojure on several small benchmark programs. The goal is not to claim competitiveness with JVM Clojure, but to identify the main sources of overhead in the implementation and to evaluate the effect of several implemented optimizations.

### 5.2.1 Benchmark methodology

The benchmarks compare TinyClojure with JVM Clojure. TinyClojure was tested with different configuration options, including LLVM optimization level, direct linking, and different integer-cache ranges. Each benchmark configuration was executed 15 times. The first run was discarded to reduce the influence of cold-start effects, such as process startup, module loading, and first-run compilation or initialization. The remaining 14 runs were used to compute the reported average time and standard deviation.

| Property              | Value                                          |
|-----------------------|------------------------------------------------|
| Host operating system | Windows 11                                     |
| Linux environment     | Debian under Windows Subsystem for Linux (WSL) |
| Processor             | 12th Gen Intel(R) Core(TM) i7-12700H           |
| Logical CPUs          | 20                                             |
| Memory                | 15 GiB                                         |
| C++ compiler          | g++ 12.2.0                                     |
| Java version          | OpenJDK 17.0.19                                |
| LLVM version          | 14.0.6                                         |
| Clojure version       | 1.11.1                                         |

■ **Table 5** Benchmark environment

The benchmark environment is summarized in Table 5.

The benchmark figures were generated from CSV output using a Python script with Matplotlib.

LLVM 14.0.6 is not the newest LLVM release, but it is the version used by the development environment of this project. Since the evaluation focuses on the TinyClojure runtime representation and compiler design rather than on

comparing LLVM releases, the benchmarks should be interpreted as measurements of this implementation in this environment.

The benchmarks are microbenchmarks. They are intended to expose specific sources of overhead, such as boxed arithmetic, function invocation, allocation, and list construction. They should not be interpreted as a complete performance characterization of general TinyClojure programs.

The comparison with JVM Clojure should also be interpreted with care. The JVM uses profiling and just-in-time compilation, so its performance may depend on warmup and on optimization decisions made during execution. Discarding only the first run reduces obvious cold-start effects, but it is not a full JVM warmup methodology. In typical long-running JVM programs, additional warmup could improve Clojure performance further. The measured slowdown of TinyClojure should therefore be interpreted mainly as a rough indication of scale, and possibly as a lower bound on the slowdown compared with a fully warmed-up JVM Clojure execution.

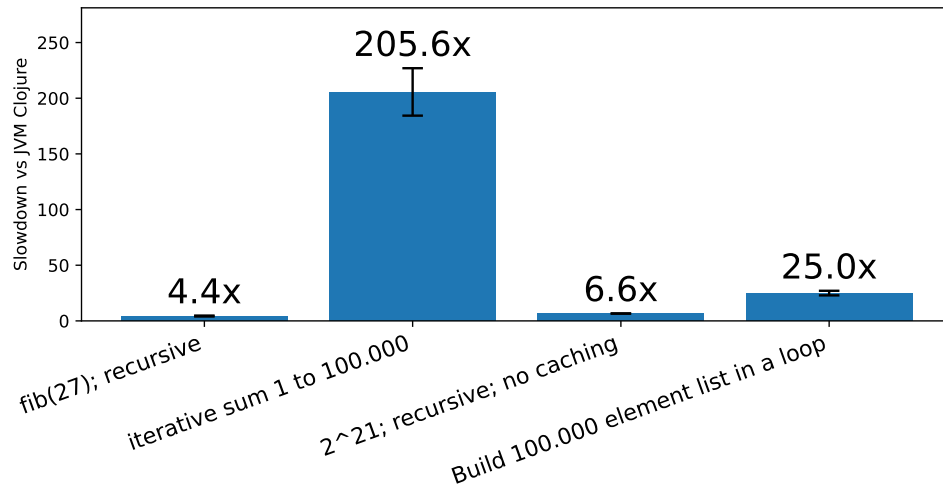
The benchmark programs were selected to exercise different parts of the implementation. The same four benchmark programs are used in the figures below; different TinyClojure configurations are compared by rerunning these programs with different options.

| Benchmark                                         | Purpose                                                             |
|---------------------------------------------------|---------------------------------------------------------------------|
| <code>fib(27); recursive</code>                   | recursive function calls and boxed integer arithmetic               |
| <code>2^21; recursive; no caching</code>          | branching recursion and repeated production of small integer values |
| <code>Build 100.000 element list in a loop</code> | allocation of list cells and boxed values                           |
| <code>iterative sum 1 to 100.000</code>           | loop-based integer addition over a growing range of values          |

■ **Table 6** Benchmark programs used in the performance evaluation.

The benchmark names in the figures correspond to the names listed in Table 6.

### 5.2.2 Overall benchmark results



■ **Figure 3** TinyClojure slowdown relative to JVM Clojure using the default optimized TinyClojure configuration. Bars show slowdown factors computed from average runtimes; error bars show standard deviation converted to slowdown units.

Figure 3 summarizes TinyClojure performance relative to JVM Clojure using the default optimized TinyClojure configuration used for the main comparison: direct linking enabled, LLVM optimization level 02, and the default integer-cache range -128:127. For each benchmark, the TinyClojure average runtime is divided by the corresponding JVM Clojure average runtime.

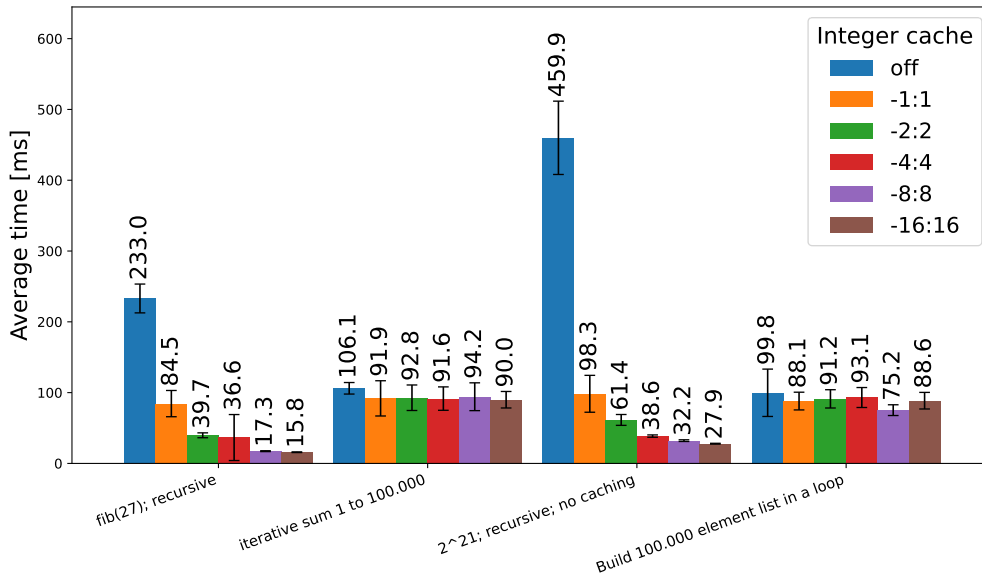
This figure shows one fixed TinyClojure configuration, not the best result selected from multiple configurations. The effects of individual configuration options are shown separately in Figure 4, Figure 5, and Figure 6.

The slowdown is not uniform across benchmarks. The branching benchmark and the Fibonacci benchmark repeatedly produce small integer values, so they benefit from integer caching. In contrast, the iterative-sum benchmark quickly produces values outside the default cache range, and the list-construction benchmark remains dominated by list-cell allocation. These differences explain why the measured slowdown varies substantially between benchmarks.

The iterative-sum benchmark is especially slow compared to JVM Clojure. The benchmark repeatedly performs integer addition, but most produced integer values lie outside the tested cache ranges. TinyClojure therefore allocates many boxed integer objects. In contrast, the JVM can often optimize numeric loops using techniques such as unboxing and scalar replacement. TinyClojure currently performs no such optimization; arithmetic is implemented through generic runtime operations on boxed values.

The list-construction benchmark stresses a different part of the runtime. Even when integer caching avoids some integer allocations, the program still has to allocate list cells. This keeps the benchmark allocation-heavy and limits the benefit of the integer cache.

### 5.2.3 Effect of integer caching



**Figure 4** Effect of integer-cache range on benchmark runtime. Bars show average runtime; error bars show standard deviation.

Figure 4 shows the effect of changing the integer-cache range. The cache has the largest effect on benchmarks that repeatedly produce small integer values. In these cases, increasing the cache range allows more results to be represented by existing runtime objects instead of newly allocated objects.

The effect is more limited when the benchmark mostly produces values outside the cache range. This is important because it shows that integer caching is a local optimization, not a general solution to the cost of boxed arithmetic. It reduces allocation pressure for common small values, but it does not remove the overhead of dynamic type checks, boxed representation, or calls to runtime helper functions.

The branching benchmark illustrates why the cache range matters. The benchmark computes  $2^{21}$  using a recursive binary structure. The intermediate results are powers of two: 1, 2, 4, 8, 16, and so on. The smallest values are produced most frequently. The value 1 is produced at the leaves, 2 one level above the leaves, and each larger power of two is produced half as often as the previous one.

For example, in a simplified model that counts only produced integer results, the value 1 is produced  $2^{21}$  times, 2 is produced  $2^{20}$  times, 4 is produced  $2^{19}$  times, and 16 is produced  $2^{17}$  times. If the integer cache covers the range  $[-16, 16]$ , all these values can be represented by cached objects.

With such a cache, the first result outside the cache is 32. The number of produced result values outside the cache is therefore

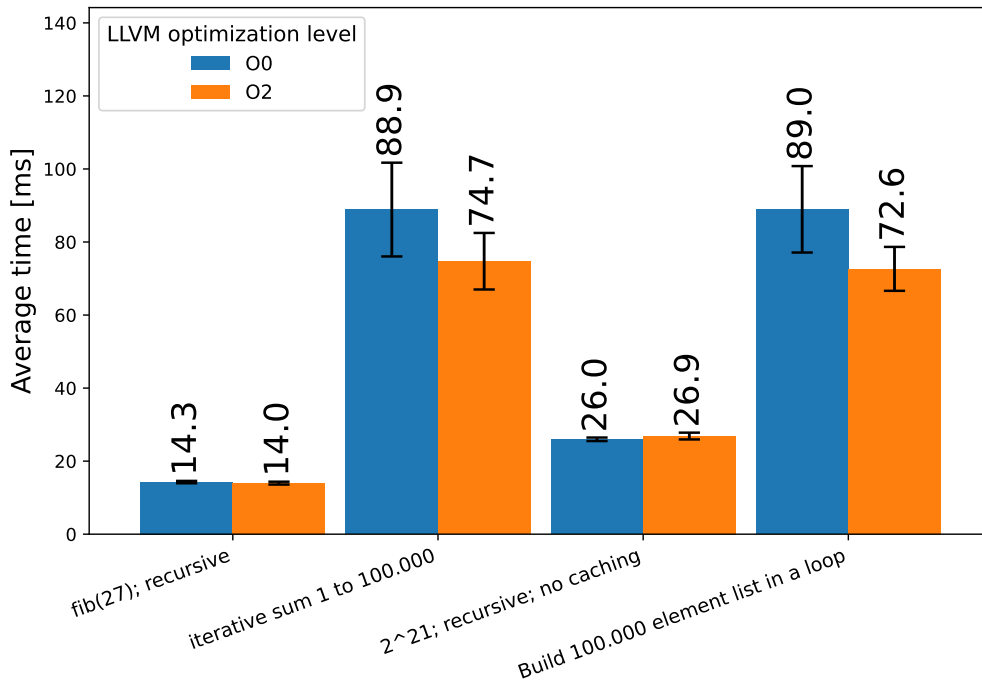
$$2^{16} + 2^{15} + \dots + 2^0 = 2^{17} - 1$$

Without caching, the same simplified model may produce up to

$$2^{21} + 2^{20} + \dots + 2^0 = 2^{22} - 1$$

integer result objects. The exact number of allocations in the implementation may differ because runtime helper functions can allocate additional temporary values, but the model explains the dominant effect: caching removes allocations for the values that occur most frequently.

#### 5.2.4 Effects of LLVM optimizations and direct linking

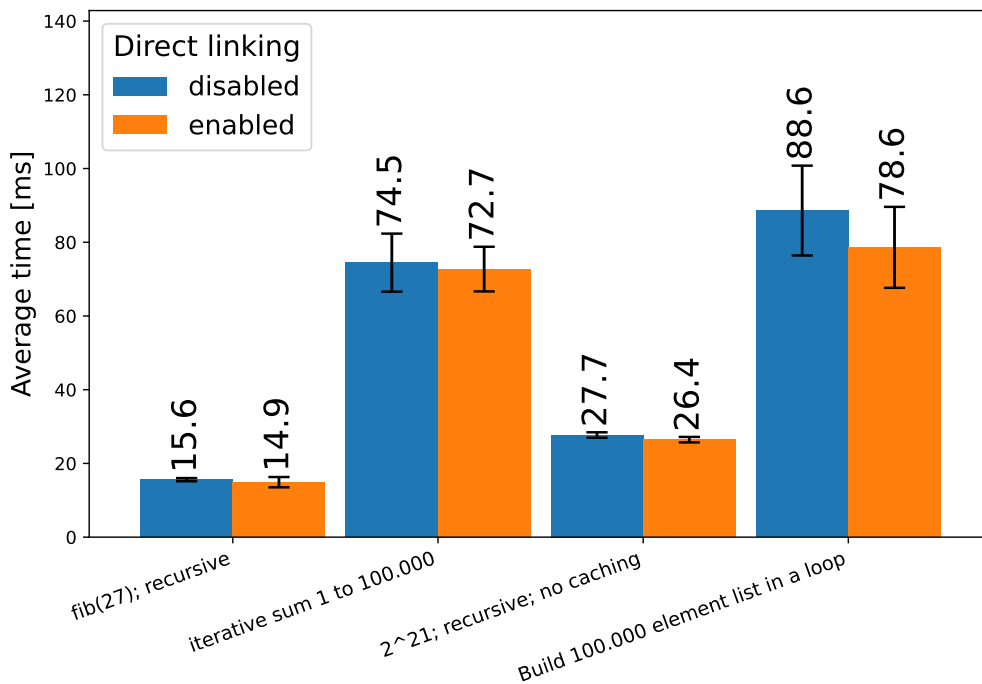


**Figure 5** Effect of LLVM optimization level on benchmark runtime. Bars show average runtime; error bars show standard deviation.

Figure 5 compares TinyClojure code generated with LLVM optimizations disabled and enabled. The effect is small and inconsistent in the measured

benchmarks. The iterative benchmarks improve slightly with optimizations enabled, while the effect on recursive benchmarks is negligible.

This result is expected for the current implementation. The generated LLVM IR mostly manipulates pointers to boxed runtime objects and calls C++ runtime helper functions. LLVM can optimize local control flow and low-level instructions, but it has limited visibility into the semantics of TinyClojure arithmetic, allocation, dynamic dispatch, and list operations. Therefore, enabling LLVM optimizations does not remove the main sources of overhead in these benchmarks.



■ **Figure 6** Effect of direct linking on benchmark runtime. Bars show average runtime; error bars show standard deviation.

Figure 6 compares ordinary dynamic invocation with direct linking. Direct linking did not show a consistent improvement in the measured benchmarks. This should not be interpreted as evidence that direct linking is inherently harmful. Rather, it suggests that the dynamic call path removed by direct linking was not the dominant cost in these particular programs.

### 5.2.5 Discussion

Compared with integer caching, direct linking and LLVM optimization level had a much smaller effect in the measured benchmarks. Direct linking removes part of the overhead of calls whose target is known during compilation, but

much of the remaining cost is elsewhere: boxed arithmetic, allocation, runtime type checks, list manipulation, and calls to runtime helper functions.

Similarly, LLVM optimization level has limited visibility into the high-level semantics of TinyClojure programs. The generated LLVM IR mostly manipulates `Object*` values and calls C++ runtime functions. LLVM can optimize local control flow and low-level instructions, but it cannot easily eliminate the overhead of boxed values or dynamic dispatch without additional compiler analysis.

Overall, the performance results show that TinyClojure executes the benchmark programs correctly, but its performance is limited by the simplicity of the runtime representation. Integer caching is the most effective measured optimization because it directly reduces allocation of frequently used boxed values. Larger improvements would likely require deeper changes, such as unboxed numeric representations, escape analysis, more precise garbage-collection integration, or more specialized code generation for statically known operations.

### 5.3 Expressiveness and limitations

The implemented TinyClojure subset is expressive enough to demonstrate the main features targeted by this thesis. Programs can define global Vars, define and call functions, use multiple arities and variadic arities, create closures, use local bindings, branch using conditionals, express loops using `loop*` and `recur`, define simple macros, and split code into loadable modules.

The end-to-end tests demonstrate that these features can be combined in ordinary programs. For example, functions can return closures, closures can be stored in Vars, Vars can be redefined, recursive functions can use `recur`, and macros from the core library can expand into forms compiled by the rest of the pipeline.

At the same time, TinyClojure is intentionally much smaller than JVM Clojure. It does not implement namespaces, Java interoperability, exceptions, protocols, persistent vectors, maps, sets, keywords, destructuring, lazy sequences, concurrency primitives, or the full Clojure standard library. Lists are the only compound data structure implemented by the language subset.

These limitations affect both expressiveness and performance. Missing data structures restrict the kinds of idiomatic Clojure programs that can be written directly. The boxed runtime representation and generic helper calls make the implementation simple and uniform, but they also introduce allocation and dispatch overhead. The simple garbage collector is sufficient for the tested programs, but it is not suitable for long-running allocation-heavy programs without more precise root tracking or more frequent safe collection points.

Overall, the evaluation shows that the implemented subset is sufficient to demonstrate the core design problem of this thesis: compiling a small dynamically typed Lisp-like language to LLVM IR requires not only code generation, but also an explicit runtime system for values, calls, Vars, closures, modules, and memory management.

## 5.4 Future work

The evaluation shows that TinyClojure implements the selected language subset, but also exposes several limitations of the current design. Some limitations are performance-related, especially the overhead of boxed values and allocation. Others are language-level limitations caused by the intentionally small subset of Clojure implemented in this thesis.

The future-work directions discussed below are summarized in Table 7. The list is not intended to be exhaustive; it focuses on extensions that follow directly from the limitations observed during evaluation.

| Direction                     | Affected part                     | Motivation                                            |
|-------------------------------|-----------------------------------|-------------------------------------------------------|
| Tagged or unboxed values      | runtime and IR generation         | reduce allocation and boxed arithmetic overhead       |
| Improved garbage collection   | runtime and generated code        | allow collection during long-running computations     |
| Direct-linking specialization | semantic analysis and calls       | make direct linking useful for known operations       |
| Additional data structures    | reader, runtime, GC, core library | support more idiomatic Clojure-like programs          |
| Destructuring                 | reader and semantic analysis      | make function parameters and bindings more expressive |
| Exceptions                    | runtime and control flow          | support structured error handling                     |

■ **Table 7** Overview of possible future-work directions.

### 5.4.1 Possible optimizations

The performance evaluation suggests that the largest measured overheads come from boxed runtime values, allocation, and runtime helper calls. The simple garbage collector is also a likely limitation for larger allocation-heavy programs, although its isolated performance impact was not measured sepa-

rately. Future optimizations should therefore focus on reducing allocations and giving the compiler more opportunities to generate specialized code.

**Reducing boxed numeric overhead.** TinyClojure currently represents most values as pointers to runtime objects. This is simple, but it means that numeric loops allocate many temporary integer objects. A future compiler could keep selected values unboxed when their type is known and box the result only when it crosses a dynamic boundary.

For example, a loop such as:

```
(loop* (i 0
        acc 0)
  (if (= i n)
    acc
    (recur (+ i 1)
           (+ acc i))))
```

is currently compiled as operations on boxed runtime objects. Each arithmetic operation goes through runtime helper functions and may allocate new boxed integer values.

A future implementation could treat this loop internally more like the following schematic representation:

```
i64 i = 0;
i64 acc = 0;

while (i != n) {
  i64 next_i = i + 1;
  i64 next_acc = acc + i;

  i = next_i;
  acc = next_acc;
}

return box_integer(acc);
```

This is not valid TinyClojure output code, but it illustrates the optimization goal: keep values such as `i` and `acc` unboxed inside the loop, and allocate a boxed result only when the value crosses back into the dynamic runtime representation.

A simpler related optimization would be to add specialized array-like storage for numeric values. The collection object itself would still be represented as an ordinary boxed runtime object, but its elements could be stored unboxed internally when they are known to be integers or floating-point values. This would not remove boxing from all numeric code, but it could reduce allocation and indirection in numerically heavy programs that operate over homogeneous numeric collections.

**Tagged values.** Another possible approach is to use tagged values. Instead of storing every value as a pointer to a heap-allocated `Object`, selected small values could be stored directly in a machine word together with a tag. This is possible because aligned heap pointers usually leave some low-order bits unused; these bits can encode whether the word represents a pointer, an integer, or another immediate value. This technique is commonly called pointer tagging [27].

A related technique is NaN-boxing, named after *not a number* floating-point values. It uses unused payload bits in IEEE 754 NaN values to store type tags and non-floating-point values inside one machine word. This representation is useful for dynamic languages whose value representation must efficiently handle both floating-point numbers and heap objects [28].

For TinyClojure, either approach could reduce allocation of small values such as integers, booleans, characters, or `nil`. The current implementation uses heap-allocated boxed runtime objects for most values, so even simple numeric operations can allocate temporary objects. Tagged values would make the runtime representation more complex, but could reduce both allocation pressure and indirection.

**Improved garbage collection.** The current collector is a simple mark-and-sweep collector that runs only at coarse-grained safe points. This is sufficient for the tested programs, but allocation-heavy computations may retain temporary objects until the current top-level evaluation finishes. A future implementation could add more precise root tracking, stack maps, or additional safe points so that collection can run safely during longer computations.

**Direct linking and specialization.** Direct linking currently removes part of the dynamic call path, but the benchmarks show that this alone is not enough to remove the dominant costs. A more useful version of direct linking could be combined with specialization. For example, direct calls to known arithmetic functions could be compiled to specialized integer operations when operand types are known.

## 5.4.2 Adding more features

TinyClojure deliberately implements only a small Clojure-inspired subset. A natural direction for future work is to add more language features and runtime value types.

### 5.4.2.1 More data structures

The most important missing feature is support for additional data structures. Lists are currently the only compound data structure implemented by the

language subset. Adding vectors, maps, sets, and keywords would make TinyClojure programs much closer to ordinary Clojure programs. This would require changes in the reader, runtime value representation, printer, equality operations, garbage collector, and core library.

#### 5.4.2.2 Destructuring

Destructuring is another useful feature, but adding it to TinyClojure is not completely straightforward. In Clojure, function parameter lists and binding forms use vectors, while lists are ordinary data. This distinction avoids ambiguity. TinyClojure currently uses lists for parameter lists and binding forms, so a form such as

```
(defn foo ((x))  
  x)
```

would be ambiguous. It could either be interpreted as a function with one destructured argument, or as a multi-arity function definition whose single arity has parameter list `(x)`. Adding destructuring would therefore likely require adding vector syntax first, or otherwise changing the syntax of function definitions and binding forms.

Clojure avoids this ambiguity because parameter lists and destructuring forms are written using vectors:

```
(defn foo [[x]]  
  x)
```

Here, the outer vector is the function parameter list and the inner vector is the destructuring pattern for the first argument. TinyClojure currently has no vector syntax, so it cannot express the same distinction directly.

#### 5.4.2.3 Better module system

A larger module and namespace system would also be useful. The current module system is intentionally simple and focuses on loading source files or AOT artifacts. A more complete system could add namespace declarations, qualified symbols, aliases, and better separation between modules. This would make larger TinyClojure programs easier to organize.

#### 5.4.2.4 Exception handling

Exception handling is another possible extension. The current implementation reports many errors directly through runtime error functions. Supporting exceptions would require a language-level exception model, runtime representation of exception objects, and integration with LLVM exception-handling

mechanisms or another control-flow strategy. This would also make it possible to implement constructs similar to `try`, `catch`, and `finally`.

For example, a future TinyClojure version could support forms such as:

```
(try
  (/ 1 x)
  (catch e
    (println "An error occurred.")))
```

Supporting such a form would require more than adding syntax. The runtime would need exception objects, and the compiler would need to generate control flow that can leave one expression and resume execution in a matching handler.

#### 5.4.2.5 Better standard library

Finally, the core library could be expanded. Many ordinary Clojure programs rely not only on language syntax, but also on a rich set of sequence, collection, string, and arithmetic functions. Extending the core library would improve the practical expressiveness of TinyClojure even without changing the compiler substantially.

These directions all point to the same trade-off. TinyClojure keeps the compiler and runtime simple enough for this thesis, but this simplicity limits both performance and language coverage. Improving performance would mostly require more specialized representations and analyses, while improving expressiveness would require a larger runtime library, more value types, and more complete language constructs.

# Conclusion

This thesis presented TinyClojure, a compiler for a restricted Clojure-inspired language targeting LLVM IR. The goal was not to implement full Clojure, but to explore what is required to compile a small dynamically typed Lisp-like language directly to LLVM instead of relying on the JVM runtime.

The implemented language subset supports literals, symbols, lists, Vars, first-class functions, closures, conditionals, recursion using `loop*` and `recur`, a simplified macro system, and loadable modules. The compiler consists of a reader, macroexpansion, semantic analysis, function compilation, and LLVM IR generation. The implementation also supports both just-in-time execution and ahead-of-time compilation of modules that can later be loaded by the TinyClojure runtime.

A substantial part of the work was the design of the runtime system. Because LLVM is not a managed language runtime, TinyClojure has to represent dynamic language features explicitly. The runtime therefore provides boxed runtime values, method tables, a uniform function calling convention, function objects and closures, global Vars, module-loading support, and a simple mark-and-sweep garbage collector. This shows that targeting LLVM directly does not remove the need for a runtime system; it only changes which parts must be implemented by the language itself.

The evaluation showed that the implemented subset behaves consistently on the tested cases. The correctness evaluation used reader unit tests, an end-to-end comparison with JVM Clojure where the semantics overlap, expected-failure tests, and a targeted AOT loading test. All automated tests written for the implementation passed at the time of evaluation. These tests do not constitute a complete Clojure conformance suite, but they provide regression coverage for the implemented TinyClojure subset.

The performance evaluation showed that TinyClojure is significantly slower than JVM Clojure on the selected microbenchmarks. The main sources of overhead are the uniform boxed value representation, allocation of temporary runtime objects, runtime helper calls, and generic dynamic operations. Integer caching was the clearest measured optimization, because it reduces allocation of frequently produced small integer values. LLVM optimization level and direct linking had smaller and less consistent effects in the measured

benchmarks, suggesting that the dominant costs are mostly in the runtime representation rather than in local LLVM IR optimization alone.

The implemented system is expressive enough to demonstrate the main features targeted by the thesis: dynamically typed values, Lisp-style forms, first-class functions, closures, macros, recursion, Vars, and module loading. At the same time, TinyClojure remains intentionally limited. It does not implement many features of JVM Clojure, such as namespaces, persistent collections, keywords, destructuring, exceptions, interoperability with the host platform, concurrency primitives, or a full standard library.

Future work can therefore proceed in two main directions. The first direction is performance. Possible improvements include reducing boxed numeric overhead, introducing tagged or unboxed value representations, improving garbage collection, adding more precise root tracking, and combining direct linking with specialization. The second direction is language coverage. Adding more data structures, destructuring, exceptions, a larger module or namespace system, and a more complete core library would make TinyClojure closer to practical Clojure-like programming.

Overall, the thesis demonstrates that compiling a small dynamically typed Lisp-like language to LLVM IR is feasible, but that the main challenge is not only generating LLVM instructions. A useful implementation also needs an explicit runtime system that provides the high-level services expected by the source language. TinyClojure illustrates these design trade-offs on a small scale and provides a foundation for further experiments with Clojure-like languages on LLVM.

# Bibliography

- [1] C. Queinnec, *Lisp in small pieces*. Cambridge University Press, 2003.
- [2] Clojure, “The Reader.” Available: <https://clojure.org/reference/reader>. [Accessed: May 10, 2026]
- [3] C. Okasaki, *Purely functional data structures*. Cambridge University Press, 1998.
- [4] Oracle Corporation, “The Java® Virtual Machine Specification,” 2024. Available: <https://docs.oracle.com/javase/specs/jvms/se12/html/index.html>
- [5] V. Alfred, S. Monica, S. Ravi, U. Jeffrey D, and others, *Compilers Principles, Techniques*. Pearson, 2007.
- [6] S. Muchnick, *Advanced compiler design implementation*. Morgan kaufmann, 1997.
- [7] Clojure Team, “The Clojure Programming Language,” 2026. Available: <https://clojure.org/>. [Accessed: May 15, 2026]
- [8] Clojure Team, “Vars and the Global Environment,” 2026. Available: <https://clojure.org/reference/vars>. [Accessed: May 15, 2026]
- [9] P. Padlewski, K. Pszeniczny, and R. Smith, “Modeling the Invariance of Virtual Pointers in LLVM,” *arXiv preprint arXiv:2003.04228*, 2020.
- [10] B. Saleil and M. Feeley, “Interprocedural specialization of higher-order dynamic languages without static analysis,” in *31st European Conference on Object-Oriented Programming (ECOOP 2017)*, 2017, pp. 23–21.
- [11] L. P. Deutsch and A. M. Schiffman, “Efficient implementation of the Smalltalk-80 system,” in *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1984, pp. 297–302.
- [12] <https://jank-lang.org/>, “The jank programming language,” 2026.
- [13] C. Lattner and V. Adve, “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation,” in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, in CGO '04. Palo Alto, California: IEEE Computer Society, 2004, p. 75.
- [14] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, “Efficiently computing static single assignment form and the control dependence graph,” *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 4,

- pp. 451–490, Oct. 1991, doi: 10.1145/115372.115320. Available: <https://doi.org/10.1145/115372.115320>
- [15] M. Serrano and P. Weis, “Bigloo: a portable and optimizing compiler for strict functional languages,” in *Static Analysis*, A. Mycroft, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 366–381.
  - [16] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, and K. Smith, “Cython: The Best of Both Worlds,” *Computing in Science & Engineering*, vol. 13, no. 2, pp. 31–39, 2011, doi: 10.1109/MCSE.2010.118
  - [17] “IEEE Standard for Floating-Point Arithmetic,” vol. , no. . pp. 1–84, 2019. doi: 10.1109/IEEESTD.2019.8766229
  - [18] “Reference to ASCII Table of Windows-1252,” 2026. Available: <https://www.ascii-code.com/>. [Accessed: May 15, 2026]
  - [19] Oracle, “Integer.valueOf(int).” Available: [https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Integer.html#valueOf\(int\)](https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Integer.html#valueOf(int)). [Accessed: May 12, 2026]
  - [20] Python Software Foundation, “Integer Objects — Python/C API Reference Manual.” Available: [https://docs.python.org/3.12/c-api/long.html#c.PyLong\\_FromLong](https://docs.python.org/3.12/c-api/long.html#c.PyLong_FromLong). [Accessed: May 13, 2026]
  - [21] R. Jones, A. Hosking, and E. Moss, *The garbage collection handbook: the art of automatic memory management*. Chapman, Hall/CRC, 2023.
  - [22] Boehm-Demers-Weiser GC project, “A garbage collector for C and C+.” Available: <https://www.hboehm.info/gc/>. [Accessed: May 13, 2026]
  - [23] MMTk project, “Memory Management Toolkit.” Available: <https://www.mmtk.io/>. [Accessed: May 13, 2026]
  - [24] Jeaye Wilkerson, “jank is now running on LLVM IR.” Available: <https://jank-lang.org/blog/2024-11-29-llvm-ir/>. [Accessed: May 13, 2026]
  - [25] Jeaye Wilkerson, “jank development update - Object model results.” Available: <https://jank-lang.org/blog/2023-08-26-object-model/>. [Accessed: May 13, 2026]
  - [26] M. Borkent, “Babashka,” 2026. Available: <https://babashka.org/>. [Accessed: May 15, 2026]
  - [27] S. Marlow, A. R. Yakushev, and S. Peyton Jones, “Faster laziness using dynamic pointer tagging,” in *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, in ICFP '07. Freiburg, Germany: Association for Computing Machinery, 2007, pp. 277–288. doi: 10.1145/1291151.1291194. Available: <https://doi.org/10.1145/1291151.1291194>
  - [28] R. Nystrom, “Optimization.” Available: <https://craftinginterpreters.com/optimization.html>. [Accessed: May 14, 2026]

# Official KOS documents

For completeness, below is the Czech assignment and declaration generated from the KOS system.

## I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Šuráň** Jméno: **Jan** Osobní číslo: **515317**  
Fakulta/ústav: **Fakulta informačních technologií**  
Zadávající katedra/ústav: **Katedra teoretické informatiky**  
Studijní program: **Informatika**  
Specializace: **Teoretická informatika**

## II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

**Kompilace podmnožiny Clojure pomocí LLVM**

Název bakalářské práce anglicky:

**Compiling a subset of Clojure with LLVM**

Jméno a pracoviště vedoucí(ho) bakalářské práce:

**Pierre Donat-Bouillud, Ph.D. katedra teoretické informatiky FIT**

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **10.03.2026**

Termín odevzdání bakalářské práce: **15.05.2026**

podpis vedoucí(ho) ústavu/katedry

podpis děkana(ky)

## III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

\_\_\_\_\_  
Datum převzetí zadání

\_\_\_\_\_  
Šuráň Jan  
Podpis studenta

## I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Šuráň** Jméno: **Jan** Osobní číslo: **515317**  
Fakulta/ústav: **Fakulta informačních technologií**  
Zadávající katedra/ústav: **Katedra teoretické informatiky**  
Studijní program: **Informatika**  
Specializace: **Teoretická informatika**

## II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

**Kompilace podmnožiny Clojure pomocí LLVM**

Název bakalářské práce anglicky:

**Compiling a subset of Clojure with LLVM**

Pokyny pro vypracování:

Clojure is a modern dialect of Lisp that runs on the JVM and supports dynamic typing, immutable data structures, first-class functions, and a macro system. While Clojure benefits from the JVM ecosystem, it is tightly coupled to its runtime and compilation model. LLVM is a modular compiler framework that enables native code generation for multiple architectures through its intermediate representation (LLVM IR). It is widely used to build optimizing compilers for modern languages.

The goal of this thesis is to define and implement a dynamically-typed subset of Clojure and compile it directly to LLVM IR. The subset should be small enough to be implementable within a bachelor thesis, yet expressive enough to support core Lisp features.

1. Research Clojure and LLVM
2. Define a subset of Clojure. The subset should include numbers, strings, lists, first-class functions, simple control flow (conditionals and recursion), and a simplified macro system
3. Implement a compiler from this subset to LLVM IR
4. Design a runtime system: implement memory management and runtime support for dynamic typing, lists, and closures.
5. Evaluate the system: assess correctness, expressiveness, and performance compared to JVM-based Clojure.

[1] Clojure documentation: <https://clojure.org>

[2] LLVM documentation: <https://llvm.org>

[3] Queinnec, C. Lisp in Small Pieces

[4] Appel, A. W. Modern Compiler Implementation

[5] <https://bernsteinbear.com/blog/compiling-a-lisp-0/>

Seznam doporučené literatury:

## DECLARATION

I, the undersigned

Student's surname, given name(s): Šuráň Jan  
Personal number: 515317  
Programme name: Informatics

hereby declare that the Bachelor's Thesis titled

Compiling a subset of Clojure with LLVM

is my own independent work and that I have declared all sources of information used in accordance with the Methodological Guideline for adhering to ethical principles when elaborating an academic final thesis and the Framework Rules for the use of artificial intelligence at CTU for study and pedagogical purposes in Bachelor and continuing Masters studies.

I declare that I have used artificial intelligence tools during the preparation and writing of the final thesis.  
I've verified the generated content. I confirm that I am aware that I am fully responsible for the content of the final thesis.

In Prague on 15.05.2026

Jan Šuráň

.....  
student's signature

# Contents of the attachment

|                                    |                                         |
|------------------------------------|-----------------------------------------|
| └ bench/ .....                     | benchmark scripts, results, and figures |
| └ └ figures/ .....                 | generated benchmark figures             |
| └ └ test-runs/ .....               | benchmark programs                      |
| └ └ clj-bench-predefs.clj .....    | JVM Clojure setup                       |
| └ └ common-bench-predefs.clj ..... | shared benchmark setup                  |
| └ └ run-benchmark.bb .....         | benchmark runner                        |
| └ └ └ visualize.py .....           | benchmark plotting script               |
| └ clj/ .....                       | TinyClojure core library                |
| └ └ core.clj .....                 | core library source                     |
| └ test/ .....                      | automated tests                         |
| └ └ aot/ .....                     | AOT and macro tests                     |
| └ └ error-files/ .....             | expected-failure inputs                 |
| └ └ CMakeLists.txt .....           | test CMake configuration                |
| └ └ forms.clj .....                | end-to-end test input                   |
| └ └ test.bb .....                  | test runner                             |
| └ └ └ test-reader.cpp .....        | reader unit tests                       |
| └ tinyclj/ .....                   | compiler and runtime source             |
| └ └ compiler/ .....                | analyzer and code generation            |
| └ └ reader/ .....                  | reader implementation                   |
| └ └ runtime/ .....                 | runtime and memory manager              |
| └ └ └ types/ .....                 | runtime value types                     |
| └ CMakeLists.txt .....             | project CMake configuration             |
| └ LICENSE .....                    | Eclipse Public License 2.0              |
| └ main.cpp .....                   | executable entry point                  |
| └ README.md .....                  | build, run, and test instructions       |
| └ thesis.pdf .....                 | thesis in PDF format                    |
| └ thesis-source/ .....             | Typst source files of the thesis        |
| └ └ thesis-attachments/ .....      | assignment and declaration documents    |