HQL: The Last Programming Language

Seoksun Jang
seoksun.jang@hlvm.dev
https://hlvm.dev

Abstract

A purely functional Lisp dialect that seamlessly integrates with the JavaScript ecosystem while providing a minimalist, expressive core. Higher Level Query LISP (HQL) combines Lisp's metaprogramming power with modern language features and full compatibility with JavaScript workflows. This paper presents the design and implementation of HQL, including its core syntax, macro system, and interoperability features. We demonstrate HQL's capabilities through elegant recursion and examples of practical programming. We introduce HQL's syntax and semantics, followed by discussions of advanced features such as macro expansion, the transpilation pipeline to JavaScript, integration with the High-Level Virtual Machine (HLVM), and module system enabling import/export from NPM and HTTP sources. Through this exposition, we show how HQL bridges the gap between Lisp's elegance and JavaScript's practicality.

1 Introduction

The programming language **Higher Level Query LISP (HQL)** is a Lisp dialect designed to bridge the gap between the timeless elegance of Lisp and the practical needs of modern software development. HQL's design follows a philosophy of a minimal core language with powerful macro facilities, enabling higher-level constructs to be built atop a simple foundation. All HQL code is composed of S-expressions (symbolic expressions), maintaining Lisp's homoiconicity (the program structure is identical to its data structure) while embracing the tooling and ecosystem of JavaScript (**?**).

HQL aspires to be "the last programming language" by embracing several core principles: "macro everywhere, minimal-core, expression-oriented, single-bundled-output, platform agnostic." This philosophy guides both the design and implementation of HQL, creating a language that combines the best aspects of functional programming with practical access to the modern JavaScript ecosystem.

HQL compiles to JavaScript, making it instantly deployable in web and server environments. It allows direct interoperability with the vast JavaScript ecosystem, including the ability to import Node.js modules, NPM packages, JSR modules, or even remote modules over HTTP. Despite this outward compatibility, HQL retains a purely functional core with first-class functions, immutable data by default, and explicit mutable state when needed. This combination makes HQL suitable for both writing concise scripts and building large-scale applications.

To illustrate HQL's expressive power, we begin by addressing the Tower of Hanoi problem—a classic recursive algorithm that elegantly demonstrates both HQL's syntax and its ability to express complex algorithms concisely. This case study motivates the need for a language like HQL that can concisely express complex algorithms.

Following this problem demonstration, we introduce the core syntax and semantics of HQL formally. We present HQL's fundamental constructs such as variable bindings, function definitions,

control flow expressions, and data structures, drawing parallels to Lisp tradition while highlighting HQL's unique features (like named and default function parameters). After establishing the core language, the exposition transitions into a chapter-oriented format, each section acting as a "chapter" delving deeper into advanced topics.

In these chapters, we explore HQL's macro system and how macro expansion works, providing a clear explanation of the compilation pipeline from HQL source to JavaScript output. We discuss the integration with the High-Level Virtual Machine (HLVM), which broadens HQL's potential by allowing it to target other platforms and enabling cross-language interoperability beyond JavaScript. We also examine HQL's module system and package management, demonstrating how HQL modules can be imported from local files, NPM packages, or URLs, and how HQL code can be published as a library in the JavaScript ecosystem.

Throughout this paper, we maintain a formal tone. Our goal is to show that HQL, despite being practical and oriented toward real-world usage, stands on a strong theoretical footing and carries forward the rich legacy of Lisp in a modern setting.

2 The Tower of Hanoi in HQL

2.1 Problem Definition

The Tower of Hanoi is a classical problem in computer science and mathematics that elegantly illustrates the power of recursive thinking. The problem consists of three rods and n disks of different sizes stacked on one rod in decreasing order of size (largest at the bottom). The objective is to move the entire stack to another rod, obeying the following rules:

1. Only one disk can be moved at a time.
2. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack or on an empty rod.
3. No disk may be placed on top of a smaller disk.

This problem famously demonstrates the elegance of recursive solutions, as moving n disks can be decomposed into moving $n-1$ disks, then moving the largest disk, then moving the $n-1$ disks again. The Tower of Hanoi has applications in algorithm design, computational complexity, and as a teaching tool for recursion.

2.2 Recursive Solution in HQL

The Tower of Hanoi can be elegantly expressed through recursion. In HQL, we define a function that prints the steps to solve the problem by breaking it down into simpler subproblems.

```
(fn hanoi (n source auxiliary target)
  (if (= n 1)
      (print "Move disk 1 from " source " to " target)
      (do
        (hanoi (- n 1) source target auxiliary)
        (print "Move disk " n " from " source " to " target)
        (hanoi (- n 1) auxiliary source target))))
```

This implementation demonstrates HQL's clarity in expressing recursive algorithms. The function `hanoi` takes four parameters: the number of disks `n`, and the three rods `source`, `auxiliary`, and `target`. When $n = 1$, we have the base case where we simply move the disk directly. For larger n, we break the problem into three steps: move $n - 1$ disks to the auxiliary rod, move the largest disk to the target, then move the $n - 1$ disks from the auxiliary to the target.

The time complexity of this algorithm is $\Theta(2^n)$, as exactly $2^n - 1$ moves are required to solve the puzzle with n disks.

The solution demonstrates several important features of HQL:

1. **Recursive Functions**: HQL naturally supports recursion, making it ideal for algorithms like Tower of Hanoi.
2. **Conditional Logic**: The `if` expression clearly separates the base case from the recursive case.
3. **Expression-Based**: All constructs including `if` and `do` are expressions that return values.
4. **Immutable Parameters**: Function parameters in HQL are immutable by default, promoting pure functions.
5. **Function Definition**: The `fn` form is used to define functions with clear parameter lists.

2.3 Extending the Solution

We can extend our solution to simulate the movement of disks by using HQL's data structures to represent the state of the rods. This demonstrates HQL's ability to work with mutable data structures and create nested helper functions.

```
(fn simulate-hanoi (n)
  (let rods {"A": [], "B": [], "C": []})

  (loop (i n)
    (when (> i 0)
      (.push (get rods "A") i)
      (recur (- i 1))))

  (print "Initial state:" rods)

  (fn move-disk (from to)
    (let disk (.pop (get rods from)))
    (.push (get rods to) disk)
    (print "Move disk " disk " from " from " to " to))

  (fn solve (n source auxiliary target)
    (if (= n 1)
        (move-disk source target)
        (do
          (solve (- n 1) source target auxiliary)
          (move-disk source target)
          (solve (- n 1) auxiliary source target))))

  (solve n "A" "B" "C"))
```

This implementation showcases how HQL can manipulate mutable data structures while maintaining the elegance of the recursive solution. The `rods` object serves as the state container, while nested helper functions provide encapsulation.

3 Core Language Syntax and Semantics

In this section, we formally introduce the syntax and semantics of HQL's core language. The design of HQL's core draws inspiration from Lisp and Scheme, focusing on a minimal set of primitive constructs that can be composed to build higher abstractions. Every HQL construct is an expression that returns a value (expression-oriented programming), and side effects are introduced in a controlled manner.

For clarity, we divide the core features into several categories: basic expressions and literals, variable bindings, functions, control flow, and data structures. We describe each category with appropriate syntax and explain its evaluation rules.

3.1 Basic Expressions and Literals

At the heart of HQL is the **S-expression** notation. An HQL source file is a sequence of S-expressions. Each S-expression is either:

- A *literal* (numeric literal, string literal, boolean, etc.),
- A *symbol* (identifier),
- A *list* of the form `(X Y Z ...)` denoting a function or macro call (where `X` is the operator and `Y`, `Z`, ... are arguments).

Comments can be included using Lisp style `;; ...` for single-line comments. Whitespace separates tokens but is otherwise ignored.

HQL supports common literal types:

- Integer and floating-point numbers (e.g., `42`, `3.14`).
- Strings enclosed in double quotes (e.g., `"hello"`).
- Booleans: `true` and `false`.
- The special literal `nil` denoting an absence of value (analogous to `null` in JavaScript or `nil` in Lisp).

Symbols are used as variable names or to refer to functions and macros. By convention, HQL symbols are case-sensitive and typically use `kebab-case` or `snake_case` naming.

3.2 Variable Bindings: `let` and `var`

HQL introduces variables through two core binding forms: `let` for immutable bindings and `var` for mutable bindings. This distinction enforces clarity about which values can change over time.

Immutable Bindings (`let`). The syntax of a `let` binding is:

```
(let <n> <expression>)
```

This creates a new binding named `<n>` whose value is the result of evaluating `<expression>`. The binding is immutable, meaning any attempt to update it will result in a compile-time or runtime error. Under the hood, a `let` corresponds to a JavaScript `const` declaration. Moreover, if the value is a compound object (such as an array or map), HQL's transpiler will freeze the object to prevent any mutation of its contents. This effectively makes deeply immutable data structures easy to declare.

As an example:

```
(let earth-radius-km 6371)
;; earth-radius-km is a constant 6371 (km radius of Earth)
;; A subsequent (set! earth-radius-km 6400) would be illegal.
```

Mutable Bindings (`var`). The syntax of a `var` binding is similar:

```
(var <n> <expression>)
```

This form creates a new variable that can be changed later. It compiles to a JavaScript `let` (mutable binding). If the initial value is an object or array, it is *not* frozen, allowing in-place modifications of that object.

To update a `var` binding, HQL provides the `set!` special form:

```
(set! <n> <new-expression>)
```

This evaluates `<new-expression>` and updates the existing variable `<n>` to that value. The `set!` form can only be used on `var`-declared variables (or on object properties).

For example:

```
(var counter 0)
(set! counter (+ counter 1))   ;; counter is now 1
```

Within a function, parameters and local `let`/`var` definitions shadow outer bindings and exist only for the lifetime of that function invocation. HQL's lexical scoping ensures that inner bindings do not affect outer ones except through returned values or mutable state explicitly passed out.

3.3 Functions: `fn` and `fx` Definitions

HQL provides two primary function constructs: `fn` for general-purpose functions and `fx` for pure functions with stronger guarantees.

3.3.1 General-Purpose Functions (`fn`)

The general syntax for defining a function with `fn` is:

```
(fn <function-name>? (<param-list>) (<-> ReturnType>?) <body>...)
```

The `<function-name>` is optional; if provided, a global binding (implicitly `let`) is created with that name referencing the function. If no name is given, the expression defines an anonymous function (which can be immediately used or stored in a variable).

The parameter list `<param-list>` is written as a parenthesized sequence of parameters. Each parameter can optionally include a type annotation and/or a default value:

- Syntax for a parameter with type: `name: Type`.
- Syntax for a parameter with default: `name = defaultValue`.
- Both can be combined: `name: Type = defaultValue`.

If a parameter has a default value, it can be omitted when calling the function; if omitted, the default expression is used. Parameters without defaults are required unless provided via a named argument. HQL also supports a varargs parameter using the notation `& rest` as the last item in the parameter list to capture any number of additional arguments in a sequence.

The function body can consist of multiple expressions; all except the last are evaluated for their side effects, and the last expression's value is returned (similar to a `do` block in Lisp). Alternatively, explicit `return` statements can be used, but typically the last expression as result is idiomatic in Lisp style.

For example:

```
;; Define a function with defaults and type annotations
(fn greet (name: String = "world" punctuation = "!")
  (print "Hello," name punctuation))
```

This defines a function `greet` that prints a greeting. It can be called as `(greet)` (using both defaults), `(greet "Alice")` (using default punctuation), or `(greet name: "Bob" punctuation: "?")` with named arguments out of positional order if desired.

3.3.2 Pure Functions (`fx`)

The `fx` construct defines pure functions with stronger guarantees:

```
(fx multiply (x: Int = 10 y: Int = 20) (-> Int)
  (* x y))
```

Pure functions defined with `fx` have these additional constraints:

- They require complete type annotations for all parameters
- They must specify a return type using the `(-> Type)` form
- They cannot have side effects or mutate external state
- All parameters are deeply copied to prevent mutation

3.3.3 Function Calls

When calling functions, HQL allows both positional and named argument passing:

- Positional arguments: e.g., `(greet "Alice" "?")`, matched by position.

- Named arguments: e.g., `(greet punctuation: "?" name: "Alice")`, which explicitly match by name. Named arguments can be given in any order, and any omitted parameters with defaults will take their default values.
- Using a placeholder with _ to skip parameters with defaults: e.g., `(multiply _ 7)` uses the default for the first parameter but specifies the second.

3.4 Control Flow Constructs

HQL includes a small set of primitive control flow expressions. Because HQL is expression-oriented, all control flow constructs produce a value.

3.4.1 `if` Expressions

The `if` expression is the fundamental conditional. Syntax:

```
(if <condition> <then-expr> <else-expr>)
```

It evaluates `<condition>`; if the result is truthy (non-false, non-nil), it evaluates and returns `<then-expr>`, otherwise it evaluates and returns `<else-expr>`. Both branches must be provided. For example:

```
(fn abs-val (x)
  (if (< x 0) (- 0 x) x))
```

3.4.2 `cond` (Conditional Clauses)

For more complex branching, HQL provides a `cond` macro similar to Lisp:

```
(cond
  (<condition1> <expr1>)
  (<condition2> <expr2>)
  ...
  (else <exprN>))
```

Each condition is checked in order; the expression for the first true condition is evaluated and returned. The `else` clause serves as a default if all previous conditions fail. The `cond` form is syntactic sugar that nests `if` expressions.

3.4.3 Loops and Recursion

Iteration in HQL is primarily achieved through recursion, supported by the `loop` and `recur` forms for explicit tail-recursive loops. The `loop` form establishes a recursion point with initial bindings, and `recur` can be used to jump back to that point with new values, similar to a continue/goto for functional loops.

Syntax:

```
(loop (<var1> <init1> <var2> <init2> ...)
  <body-expr> ...
  (recur <expr1> <expr2> ...))
```

The `loop` sets up local variables `var1`, `var2`, ... with initial values. Within the loop body, a `recur` call can appear, which must provide exactly the same number of expressions as there were loop variables. When `recur` is executed, it jumps back to the top of the `loop`, assigning the evaluated expressions as the new values of the loop variables, and then continues execution from the top of the loop body. This is guaranteed to be a tail-call (no additional stack frame is consumed for the jump) by the HQL transpiler, enabling efficient looping without risk of stack overflow.

HQL also provides these higher-level iteration constructs:

```
;; While loop - repeat while condition is true
(while (< i 10)
  (print i)
  (set! i (+ i 1)))

;; Repeat loop - execute body n times
(repeat 3
  (print "Hello"))

;; For loop - various formats
(for (i 5)          ;; i from 0 to 4
  (print i))

(for (i 1 5)        ;; i from 1 to 4
  (print i))

(for (i 0 10 2)     ;; i from 0 to 9 by step 2
  (print i))

;; Named parameter format
(for (i from: 0 to: 10 by: 2)
  (print i))
```

3.5 Data Structures and Literal Collections

Although at its core HQL treats everything as lists (S-expressions) and symbols, it provides literal syntax for more complex data structures to facilitate practical programming:

1. **Vectors (Arrays):** Square brackets denote an array literal. E.g., `[1 2 3]` is an array of three numbers. Arrays in HQL transpile to JavaScript `Array` objects. They can be manipulated with standard JS methods (using the interop syntax) or via library functions.

2. **Hash Maps (Objects):** Curly braces denote a hash map (key-value collection). For example, `{"name" "Alice" "age" 30}` represents an object with two key-value pairs. Keys are often strings or keywords; values can be any expression. These maps transpile to JavaScript `Object` (or potentially `Map` if specified).

3. **Hash Sets:** HQL can denote a set literal using the notation `#[...]` (similar to Clojure's set literal). For instance, `#[1 2 3]` would be a set of the numbers 1,2,3. The parser recognizes the `#` prefix and constructs an appropriate set object (a JavaScript `Set`).

4. **Lists:** Since HQL itself is list-structured, creating a list (linked list structure) can be done by quoting: e.g., `'(1 2 3)` yields a list of three elements. However, HQL by default favors

vectors for sequence data since they map directly to JS arrays.

4 Macro System and Expansion

4.1 Macro Philosophy

One of HQL's core principles is "macro everywhere." This means that beyond a minimal kernel of essential forms, most language features are implemented as macros. This approach enables a tiny, focused core while providing a rich, expressive surface language. Macros in HQL transform code at compile-time, allowing developers to extend the language with new syntax and abstractions (**?**).

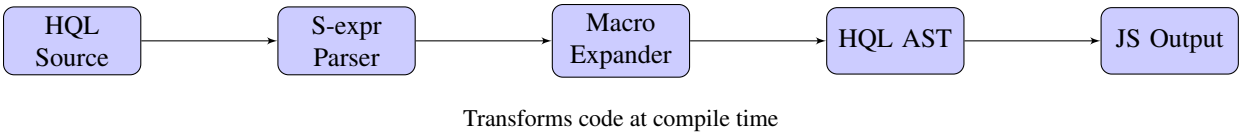

Figure 1: Macro System in HQL Compilation Pipeline

4.2 Macro Definition

HQL supports two types of macros:

1. **System-level macros** with `defmacro`, which are globally available
2. **User-level macros** with the `macro` form, which can be exported and imported like other module members

A simple macro definition looks like this:

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

This definition creates a `when` macro that executes a body of expressions only if a test condition is true. The macro uses quasiquotation (backtick) to build a template, with unquote ( ) and unquote-splicing (@) to insert values into that template.

4.3 Macro Expansion Process

The HQL macro expansion process follows these steps:

1. Parse source code into S-expressions
2. Identify macro calls in the S-expressions
3. Apply macro transformations, replacing each macro call with its expanded form
4. Repeat until no more macros can be expanded (fixed-point iteration)
5. Continue with the compilation process using fully expanded code

For example, the expansion of:

```
(when (> x 0)
  (print "Positive")
  (process x))
```

Results in:

```
(if (> x 0)
    (do
      (print "Positive")
      (process x))
    nil)
```

This powerful transformation capability allows HQL to have a minimal core while providing rich syntax, similar to the approach taken by Scheme and Clojure (**?**).

4.4 Example Macro: `let`

The `let` form for multiple bindings is implemented as a macro in HQL:

```
(defmacro let (bindings & body)
  (if (symbol? bindings)
      `(let-simple ~bindings ~@body)
      `((fn ~(map first (partition 2 bindings))
        ~@(map second (partition 2 bindings)))
        ~@body)))
```

This macro transforms a multi-binding `let` expression into an immediately invoked function, demonstrating the power of macros to extend the language with new syntactic forms.

5 Transpiler Architecture and Pipeline

The HQL transpiler is a sophisticated system that processes HQL source code through multiple stages to generate executable JavaScript. Understanding this pipeline is essential for comprehending how HQL's high-level constructs translate to efficient JavaScript.

5.1 Complete Pipeline Overview

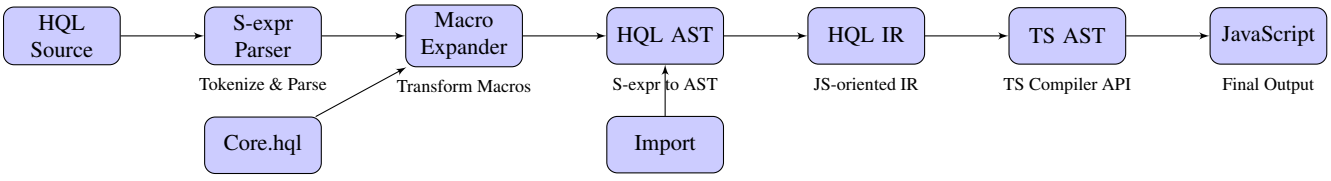

Figure 2: HQL Transpiler Pipeline Architecture

The HQL transpiler follows a multi-stage pipeline (**?**):

1. **S-Expression Parsing**: Converts HQL source code into a tree of S-expressions
2. **Import Processing**: Resolves module dependencies (local, NPM, JSR, HTTP)
3. **Macro Expansion**: Transforms code through recursive macro application
4. **HQL AST Conversion**: Converts expanded S-expressions to a structured AST
5. **HQL IR Generation**: Transforms the AST into a JavaScript-oriented intermediate representation
6. **TypeScript AST Generation**: Converts the IR to TypeScript's AST using the TypeScript Compiler API
7. **JavaScript Code Generation**: Produces the final JavaScript output

5.2 S-Expression Parsing

The parser tokenizes HQL source code and builds a tree of S-expressions. It handles various token types including lists, symbols, strings, and numbers, while tracking source positions for error reporting. The parser also recognizes special syntax like vectors `[...]`, maps `{...}`, and sets `#[...]`.

5.3 Import Processing

HQL's import system supports multiple formats and sources:

- Vector imports: `(import [symbol1, symbol2] from "./module.hql")`
- Named imports with aliases: `(import [name as alias] from "module")`
- Module imports: `(import name from "./module.hql")`
- Various sources: local files, NPM packages, JSR modules, HTTP URLs

5.4 Macro Expansion

The macro expansion phase applies transformations defined by macros to the code. It uses a fixed-point iteration algorithm that keeps applying macros until no further changes occur. This ensures that nested macros and macros that expand into other macros are fully processed. The time complexity of macro expansion is typically linear in the size of the expanded code.

5.5 Intermediate Representation and Code Generation

After macro expansion, the code undergoes several transformations:

1. Conversion to HQL AST (a structured representation of the expanded code)
2. Transformation to a JavaScript-oriented IR (mapping HQL constructs to JS equivalents)
3. Conversion to TypeScript AST using the TS Compiler API
4. Final JavaScript code generation

5.6 Runtime Functions

The transpiler includes a set of runtime functions that are prepended to the output JavaScript. These functions implement core HQL functionality in JavaScript.

```
function get(obj, key, notFound = null) {
  if

\section{JavaScript Interoperability}

One of HQL's most powerful features is its seamless JavaScript
    interoperability. This allows HQL to leverage the vast JavaScript
    ecosystem while maintaining its Lisp-like syntax and functional
    programming model.

\subsection{Property and Method Access}

HQL provides dot notation for accessing JavaScript properties and
    methods:

\begin{lstlisting}
;; Property access
(let name person.name)
(let length array.length)

;; Method calls
(let upper-message (message.toUpperCase))
(let message-parts (message.split " "))
```

5.7 Chained Method Calls

HQL supports method chaining in both traditional S-expression style and a more modern dot-chain syntax:

```
;; Traditional S-expression style
(print (.join (.sort (.map (.filter numbers (lambda (n) (> n 5)))
                           (lambda (n) (* n 2)))
                     (lambda (a b) (- a b)))
              ", "))

;; Modern dot-chain style
(print (numbers
  .filter (lambda (n) (> n 5))
  .map (lambda (n) (* n 2))
  .sort (lambda (a b) (- a b))
  .join ", "))
```

The dot-chain syntax is automatically transformed into the traditional nested S-expression form by the HQL reader/macro system, providing a more intuitive and readable syntax for method chaining.

5.8 Module Imports from JavaScript Ecosystem

HQL can import modules from various sources in the JavaScript ecosystem:

```
;; NPM package import
(import express from "npm:express")
(let app (express))
(app.use (express.json))

;; JSR module import
(import chalk from "jsr:@nothing628/chalk@1.0.0")
(console.log (chalk.green "This text is green!"))

;; HTTP import
(import path from "https://deno.land/std@0.170.0/path/mod.ts")
(let joined-path (path.join "folder" "file.txt"))
```

6 Module System

HQL's module system enables code organization and reuse with an expressive and flexible approach to imports and exports.

6.1 Export Declarations

HQL provides a vector-based syntax for exports:

```
;; Export multiple symbols
(export [add, subtract, multiply])

;; Export a single symbol
(export [add])
```

6.2 Import Declarations

The import system supports various patterns:

```
;; Named imports
(import [add, subtract, multiply] from "./math.hql")

;; Imports with aliases
(import [add as plus, subtract as minus] from "./math.hql")

;; Namespace import (entire module)
(import math from "./math.hql")
(math.add 3 4)
```

6.3 Macro Imports and Exports

A unique feature of HQL is the ability to import and export macros between modules:

```
;; In module a.hql
(macro user-log (& args)
  `(console.log "LOG:" ~@args))

(export [user-log])

;; In module b.hql
(import [user-log] from "./a.hql")
(user-log "Imported macro works!")
```

7 High-Level Virtual Machine (HLVM) Integration

7.1 What is HLVM?

The High-Level Virtual Machine (HLVM) is a runtime environment designed to support multiple high-level languages, including HQL. It provides a unified platform for cross-language interoperability, enabling code written in different languages to seamlessly work together.

7.2 HQL and HLVM Architecture

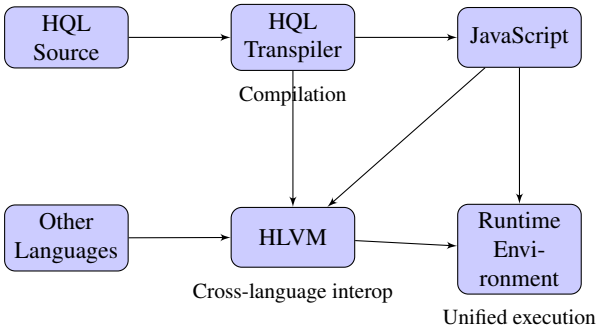

Figure 3: HQL Integration with HLVM

7.3 nREPL Integration

HQL integrates with nREPL (Network REPL) to provide a persistent programming environment:

- Allows interactive development and testing
- Provides persistent state across sessions
- Enables module-based organization within the REPL

- Supports seamless interaction with the HLVM

```
;; REPL example
hql[user]> (import [sqrt] from "math")
[1 symbol imported]

hql[user]> (sqrt 16)
4

;; Create and switch to a new module
hql[user]> :go geometry
Switched to module: geometry

hql[geometry]> (fn calculate-area (radius)
                 (* Math.PI (* radius radius)))
[Function: calculate-area]
```

8 Advanced Features

8.1 Enumerated Types

HQL provides a built-in enum syntax for defining enumerated types:

```
;; Simple enum
(enum Direction
  (case north)
  (case south)
  (case east)
  (case west)
)

;; Enum with raw values
(enum HttpStatus : Int
  (case ok 200)
  (case notFound 404)
  (case serverError 500)
)

;; Using enums with type inference
(fn process (status: HttpStatus)
  (if (= status .ok)
      "Everything is fine"
      "Something went wrong"))

(process status: .ok)  ;; Type inference allows .ok shorthand
```

8.2 Classes and Structs

HQL supports object-oriented programming with class and struct definitions:

```
;; Class definition
(class Person
  ;; Fields
  (var name)
  (var age)

  ;; Constructor
  (constructor (name age)
    (set! this.name name)
    (set! this.age age))

  ;; Method
  (fn greet ()
    (+ "Hello, " this.name)))

;; Creating an instance
(let person (new Person "Alice" 30))
(print (person.greet))  ;; "Hello, Alice"
```

8.3 Pure Functions

The `fx` construct allows defining pure functions with strong guarantees:

```
(fx add-pure (x: Int y: Int = 10) (-> Int)
  (+ x y))
```

Pure functions have several key characteristics:

- Type safety with complete type annotations
- No side effects or mutations
- Parameters are deeply copied to prevent external state changes
- Explicit return type declaration with `(-> Type)`

9 Conclusion

HQL represents a significant advancement in programming language design by successfully bridging the gap between Lisp's elegant, minimal syntax and modern JavaScript's practical ecosystem. Through its design principles of "macro everywhere, minimal-core, expression-oriented, single-bundled-output, platform agnostic," HQL delivers on the promise of being "the last programming language" a developer might need to learn.

The language's core strength lies in its ability to combine seemingly contradictory priorities:

- Functional purity with practical side effects
- Lisp's elegant S-expressions with JavaScript's vast ecosystem
- A minimal core with extensive expressivity through macros
- Local development with global accessibility through nREPL

By integrating with the High-Level Virtual Machine (HLVM), HQL positions itself not just as another language but as part of a comprehensive programming environment where developers can leverage multiple languages seamlessly. The nREPL integration further enhances this capability by providing a persistent, module-aware programming environment that evolves with the developer's needs.

HQL's transpiler pipeline demonstrates sophisticated compiler design principles, turning high-level Lisp-like code into efficient JavaScript while preserving semantics and performance. The macro system enables powerful language extensions without modifying the core compiler, allowing the language to evolve organically through its community.

In an era where programming languages often specialize in either theoretical elegance or practical utility, HQL stands out by refusing to compromise on either. It embraces the timeless wisdom of Lisp while fully participating in the modern JavaScript ecosystem, creating a bridge between functional programming's mathematical rigor and web development's practical necessities.

As JavaScript continues to dominate as the lingua franca of the web, HQL offers developers a more expressive, safer, and more powerful way to harness that ecosystem while writing code that is more maintainable and reasoning-friendly. This combination of practical utility and theoretical elegance positions HQL as not just another programming language, but potentially as "the last programming language" a developer might need to learn.

10 Appendix: Syntax Reference

10.1 Core Expressions

```
;; Variable bindings
(let x 10)
(var y 20)

;; Function definitions
(fn add (x y) (+ x y))
(fx multiply (x: Int y: Int) (-> Int) (* x y))

;; Conditional expressions
(if (> x 0) "positive" "non-positive")
(cond
  ((< x 0) "negative")
  ((= x 0) "zero")
  (else "positive"))

;; Loops
(loop (i 0)
  (when (< i 5)
    (print i)
    (recur (+ i 1))))

(while (< i 10)
  (print i)
  (set! i (+ i 1)))

(for (i 0 10 2)
  (print i))
```

10.2 Data Structures

```
;; Vector/Array
[1, 2, 3, 4, 5]

;; Hash Map/Object
{"name": "Alice", "age": 30}

;; Set
#[1, 2, 3, 4, 5]

;; List (quoted)
'(1 2 3 4 5)
```

10.3 JavaScript Interoperability

```
;; Property access
person.name
array.length

;; Method calls
(message.toUpperCase)
(array.push 42)

;; Dot-chain notation
(numbers
  .filter (lambda (n) (> n 5))
  .map (lambda (n) (* n 2))
  .join ", ")

;; Module imports
(import express from "npm:express")
(import chalk from "jsr:@nothing628/chalk@1.0.0")
(import path from "https://deno.land/std@0.170.0/path/mod.ts")
```

10.4 Module System

```
;; Exports
(export [add, subtract, multiply])

;; Imports
(import [add, subtract] from "./math.hql")
(import [add as plus] from "./math.hql")
(import math from "./math.hql")
```

10.5 Macros

```
;; Macro definition
(defmacro when (test & body)
  `(if ~test (do ~@body) nil))

;; User-level macro
(macro unless (test & body)
  `(if (not ~test) (do ~@body) nil))

;; Exporting macros
(export [unless])
```