Eshkol: A Scientific Computing Language with First-Class Automatic Differentiation

Scheme-like syntax meets LLVM performance with built-in calculus

Executive Summary

Eshkol is a modern functional programming language designed from the ground up for scientific computing and machine learning. It combines the elegance of Scheme with the performance of native code compilation via LLVM, featuring **automatic differentiation as a first-class language primitive**.

Key Differentiators

Feature	Eshkol	Python/NumPy	Julia	Scheme
Native Compilation	LLVM	Interpreted	JIT	Varies
Built-in Autodiff	**First-class**	Library (JAX)	Library	None
Vector Calculus	**Built-in operators**	Library	Library	None
Garbage Collection	**None (Arena)**	Yes	Yes	Yes
Homoiconicity	**Yes**	No	No	Yes
REPL + JIT	**Yes**	Yes	Yes	Yes

Table of Contents

Getting Started

Installation

```
# Clone and build
git clone https://github.com/tsotchke/eshkol.git
cd eshkol
mkdir build && cd build
cmake .. && make

# Run a program
./eshkol-run examples/fibonacci.esk

# Start interactive REPL
./eshkol-repl
```

Hello World

```
;; hello.esk
(display "Hello, Eshkol!")
(newline)
```

First Derivative

```
;; Compute derivative of f(x) = x² at x = 3
(define f (lambda (x) (* x x)))
(display (derivative f 3.0))  ;; → 6.0
```

Language Overview

Eshkol uses **S-expression syntax** familiar to Lisp/Scheme programmers:

```
;; Variables
(define x 42)
(define pi 3.14159)
(define name "Eshkol")

;; Functions
(define (square x) (* x x))
(define (add a b) (+ a b))

;; Lambda expressions
(define double (lambda (x) (* 2 x)))

;; Lists
(define nums (list 1 2 3 4 5))
(car nums)   ;; → 1
(cdr nums)   ;; → (2 3 4 5)

;; Conditionals
(if (> x 0) "positive" "non-positive")

(cond ((< x 0) "negative")
      ((= x 0) "zero")
      (else "positive"))

;; Let bindings
(let ((x 10) (y 20))
  (+ x y))  ;; → 30

;; Iteration
(do ((i 0 (+ i 1)))
    ((= i 10) 'done)
  (display i))
```

Core Features

Data Types

Type	Example	Description
Integer	`42` , `-17`	64-bit signed (exact)
Float	`3.14` , `2.5e-10`	IEEE 754 double (inexact)
Boolean	`#t` , `#f`	True/False
Character	`#\a` , `#\newline`	Unicode
String	`"hello"`	UTF-8
Symbol	`'foo`	Interned identifier
List	`(list 1 2 3)`	Linked cons cells
Vector	`(vector 1 2 3)`	Indexed array
Tensor	`(tensor ...)`	N-dimensional array
Lambda	`(lambda (x) x)`	First-class function
Closure	Captured lambda	Function + environment

180+ Built-in Functions

Arithmetic: `+` , `-` , `*` , `/` , `abs` , `floor` , `ceiling` , `round` , `modulo` , `remainder` , `quotient` , `gcd` , `lcm` , `min` , `max` , `expt`

Math: `sin` , `cos` , `tan` , `asin` , `acos` , `atan` , `sinh` , `cosh` , `tanh` , `exp` , `log` , `log10` , `sqrt` , `pow`

Comparison: `<` , `>` , `=` , `<=` , `>=` , `eq?` , `eqv?` , `equal?`

Lists: `cons` , `car` , `cdr` , `list` , `append` , `reverse` , `length` , `map` , `filter` , `fold` , `member` , `assoc` , + 30 compound accessors (`caar` , `cadr` , `caddr` , ...)

Strings: `string-append` , `substring` , `string-length` , `string->list` , `number->string`

Vectors: `vector` , `make-vector` , `vector-ref` , `vector-set!` , `vector-length`

I/O: `display` , `newline` , `printf` , `open-input-file` , `read-line` , `write-string`

Automatic Differentiation

The Killer Feature

Eshkol provides **three modes of automatic differentiation** as built-in language primitives:

1. Symbolic Differentiation (`diff`)

Compile-time expression transformation:

```
(diff (* x x) x)           ;; → (* 2 x)
(diff (sin (* x x)) x)     ;; → (* (cos (* x x)) (* 2 x))
(diff (+ (* x x) (* 3 x)) x)  ;; → (+ (* 2 x) 3)
```

2. Forward-Mode AD (`derivative`)

Numerical derivatives using dual numbers:

```
(define f (lambda (x) (* x x x)))  ;; f(x) = x³
(derivative f 2.0)                  ;; → 12.0 (f'(x) = 3x²)

(define g (lambda (x) (sin (* x x))))
(derivative g 1.0)                  ;; → 1.0806... (2x·cos(x²))
```

3. Reverse-Mode AD (`gradient` , `jacobian` , `hessian`)

Computation graph-based differentiation for multivariate functions:

```
;; Gradient of f(x,y) = x² + y²
(define f (lambda (v)
  (+ (* (vref v 0) (vref v 0))
     (* (vref v 1) (vref v 1)))))

(gradient f (vector 3.0 4.0))  ;; → #(6.0 8.0)

;; Jacobian matrix for vector-valued functions
(define polar->cartesian (lambda (v)
  (let ((r (vref v 0)) (theta (vref v 1)))
    (vector (* r (cos theta))
            (* r (sin theta))))))

(jacobian polar->cartesian (vector 1.0 0.0))
;; → #((1.0 0.0) (0.0 1.0))

;; Hessian matrix (second derivatives)
(hessian f (vector 1.0 1.0))  ;; → #((2.0 0.0) (0.0 2.0))
```

Vector Calculus Operators

Built-in operators for physics and engineering:

```
;; Divergence: ∇·F
(define F (lambda (v) (vector (* 2 (vref v 0)) (* 3 (vref v 1)))))
(divergence F (vector 1.0 1.0))  ;; → 5.0

;; Curl: ∇×F (3D only)
(define rotating (lambda (v)
  (vector (- 0 (vref v 1)) (vref v 0) 0.0)))
(curl rotating (vector 1.0 1.0 0.0))  ;; → #(0.0 0.0 2.0)

;; Laplacian: ∇²f
(define harmonic (lambda (v)
  (- (* (vref v 0) (vref v 0))
     (* (vref v 1) (vref v 1)))))
(laplacian harmonic (vector 1.0 1.0))  ;; → 0.0 (it's harmonic!)

;; Directional derivative: D_u f
(directional-derivative f (vector 3.0 4.0) (vector 1.0 0.0))  ;; → 6.0
```

Tensor & Matrix Operations

Creation

```
(vector 1.0 2.0 3.0)          ;; 1D vector
(zeros 3 4)                    ;; 3×4 matrix of zeros
(ones 2 3)                     ;; 2×3 matrix of ones
(eye 3)                        ;; 3×3 identity matrix
(arange 10)                    ;; #(0 1 2 3 4 5 6 7 8 9)
(linspace 0 1 5)               ;; #(0.0 0.25 0.5 0.75 1.0)
(reshape (arange 6) 2 3)       ;; 2×3 matrix
```

Operations

```
;; Element-wise arithmetic
(tensor-add v1 v2)
(tensor-sub v1 v2)
(tensor-mul v1 v2)
(tensor-div v1 v2)

;; Linear algebra
(tensor-dot v1 v2)             ;; Dot product
(matmul A B)                   ;; Matrix multiplication
(transpose M)                  ;; Transpose
(norm v)                       ;; L2 norm
(trace M)                      ;; Sum of diagonal
(outer u v)                    ;; Outer product

;; Reductions
(tensor-sum v)                 ;; Sum all elements
(tensor-mean v)                ;; Mean value
(tensor-reduce v + 0)          ;; Custom reduction
(tensor-reduce-all M * 1)      ;; Reduce entire tensor

;; Shape manipulation
(flatten M)                    ;; Flatten to 1D
(reshape v 2 3)                ;; Reshape to 2×3
(tensor-shape M)               ;; Get dimensions
```

Example: Neural Network Layer

```
(define input (vector 1.0 0.5 -0.5))
(define weights (vector 0.5 0.3 0.2))
(define bias 0.1)

;; Linear layer: y = Wx + b
(define output (+ (tensor-dot input weights) bias))
(display output)  ;; → 0.65
```

Higher-Order Functions

Map, Filter, Fold

```
;; Map: apply function to each element
(map (lambda (x) (* x 2)) (list 1 2 3 4))  ;; → (2 4 6 8)

;; Multi-list map
(map + (list 1 2 3) (list 10 20 30))  ;; → (11 22 33)

;; Filter: select matching elements
(filter (lambda (x) (> x 2)) (list 1 2 3 4 5))  ;; → (3 4 5)

;; Fold: reduce to single value
(fold + 0 (list 1 2 3 4 5))    ;; → 15
(fold * 1 (list 1 2 3 4 5))    ;; → 120
```

Closures

```
;; Factory pattern
(define (make-adder n)
  (lambda (x) (+ x n)))

(define add5 (make-adder 5))
(define add10 (make-adder 10))

(add5 3)   ;; → 8
(add10 3)  ;; → 13

;; Counter with mutable state
(define (make-counter)
  (let ((count 0))
    (lambda ()
      (set! count (+ count 1))
      count)))
```

Function Composition

```
(define (compose f g)
  (lambda (x) (f (g x))))

(define (square x) (* x x))
(define (double x) (* 2 x))

(define square-then-double (compose double square))
(define double-then-square (compose square double))

(square-then-double 3)  ;; → 18 (3² × 2)
(double-then-square 3)  ;; → 36 ((3 × 2)²)
```

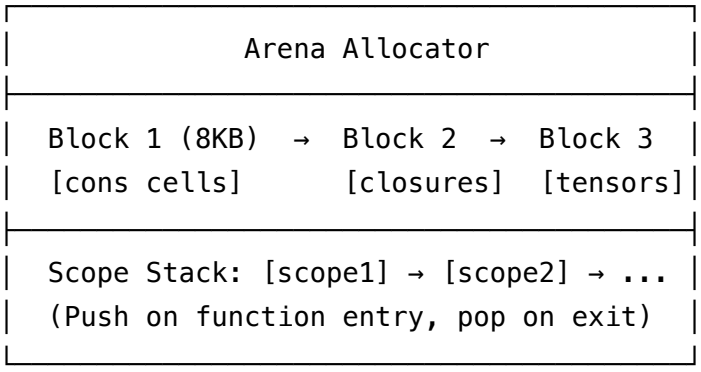
Standard Library Combinators

```
;; From lib/stdlib.esk
(identity x)           ;; Return x unchanged
(constantly 5)         ;; Returns (lambda (x) 5)
(flip f)               ;; Swap arguments: (flip -) → (lambda (a b) (- b a))
(negate pred)          ;; (negate even?) → odd?

;; Currying
(define curried-add (curry2 +))
((curried-add 5) 10)   ;; → 15

;; Partial application
(define add5 (partial2 + 5))
(add5 10)              ;; → 15
```

Memory Model

Arena Allocation (No Garbage Collection)

Eshkol uses **arena-based memory management** for deterministic, low-latency performance:



Benefits:

- **No GC pauses** - Critical for real-time applications
- **Cache-friendly** - Sequential allocation
- **Deterministic cleanup** - Memory freed when scope exits
- **Fast allocation** - O(1) bump-pointer allocation

Tagged Value System

Every runtime value is a 16-byte tagged structure:

```
struct eshkol_tagged_value {
    uint8_t type;        // INT64, DOUBLE, CONS_PTR, CLOSURE_PTR, etc.
    uint8_t flags;       // Exactness (Scheme compatibility)
    uint16_t reserved;
    union {
        int64_t int_val;
        double double_val;
        void* ptr_val;
    } data;
};
```

This enables **polymorphic operations** with runtime type dispatch:

```
(list 1 2.5 "hello" (lambda (x) x))  ;; Mixed-type list supported!
```

C Interoperability

Foreign Function Interface

Call C functions directly:

```
;; Declare external C functions
(extern void printf char* ...)
(extern void* malloc int)
(extern double sin double)

;; Use them in Eshkol
(printf "The sine of %f is %f\n" 1.0 (sin 1.0))
```

Function Aliasing

Map Eshkol names to C names:

```
;; Map friendly names to C functions
(extern void log-message :real printf char* ...)
(extern void* allocate-memory :real malloc int)

(log-message "Allocating %d bytes\n" 1024)
```

Interactive REPL

Features

- **Tab completion** for all 180+ builtins
- **Syntax highlighting** with ANSI colors
- **Command history** (persistent across sessions)
- **Multi-line input** with balanced parenthesis detection
- **JIT compilation** via LLVM ORC

REPL Commands

```
:help      Show help
:quit      Exit REPL
:env       Show defined symbols
:load FILE Load and execute file
:reload    Reload last file
:time EXPR Time expression execution
:ast EXPR  Show AST for expression
:clear     Clear screen
:history   Show command history
```

Example Session

```
$ ./eshkol-repl
Eshkol REPL v0.1.1
Type :help for assistance, :quit to exit

eshkol> (define (square x) (* x x))
square

eshkol> (square 5)
25

eshkol> (map square (list 1 2 3 4 5))
(1 4 9 16 25)

eshkol> (define f (lambda (x) (* x x x)))
f

eshkol> (derivative f 2.0)
12.0

eshkol> :quit
Goodbye!
```

Examples

Fibonacci Sequence

```
(define (fibonacci n)
  (if (< n 2)
      n
      (+ (fibonacci (- n 1))
         (fibonacci (- n 2)))))

;; Print first 10 Fibonacci numbers
(do ((i 0 (+ i 1)))
    ((= i 10) 'done)
  (display "fib(")
  (display i)
  (display ") = ")
  (display (fibonacci i))
  (newline))
```

Gradient Descent Optimization

```
;; Minimize f(x,y) = (x-3)² + (y-4)²
(define (loss v)
  (+ (* (- (vref v 0) 3.0) (- (vref v 0) 3.0))
     (* (- (vref v 1) 4.0) (- (vref v 1) 4.0))))

(define (gradient-step point lr)
  (let ((grad (gradient loss point)))
    (vector (- (vref point 0) (* lr (vref grad 0)))
            (- (vref point 1) (* lr (vref grad 1))))))

;; Optimize
(define start (vector 0.0 0.0))
(define lr 0.1)

(display "Start: ") (display start) (newline)
(display "Loss: ") (display (loss start)) (newline)

(define step1 (gradient-step start lr))
(display "After step 1: ") (display step1) (newline)
(display "Loss: ") (display (loss step1)) (newline)
;; Converges toward (3, 4)
```

Neural Network Training

```
;; Model: y = w*x + b (linear regression)
(define (model w b x) (+ (* w x) b))

;; Loss: Mean Squared Error
(define (mse w b x target)
  (let ((pred (model w b x)))
    (* (- pred target) (- pred target))))

;; Training step using autodiff
(define (train-step w b lr x y)
  (let ((grad-w (derivative (lambda (w-val) (mse w-val b x y)) w))
        (grad-b (derivative (lambda (b-val) (mse w b-val x y)) b)))
    (list (- w (* lr grad-w))
          (- b (* lr grad-b)))))

;; Train to learn y = 2x
(define training-x (list 1.0 2.0 3.0 4.0 5.0))
(define training-y (list 2.0 4.0 6.0 8.0 10.0))

;; After training: w ≈ 2.0, b ≈ 0.0
```

Merge Sort

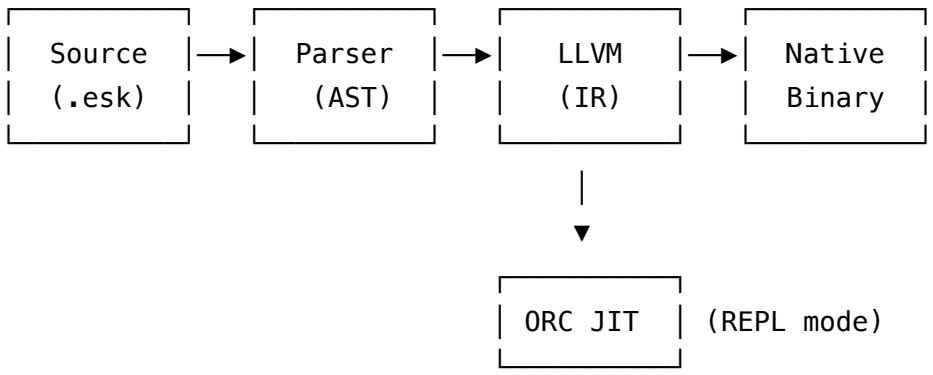
```
(define (sort lst less?)
  (define (merge l1 l2)
    (cond ((null? l1) l2)
          ((null? l2) l1)
          ((less? (car l1) (car l2))
           (cons (car l1) (merge (cdr l1) l2)))
          (else
           (cons (car l2) (merge l1 (cdr l2))))))

  (if (or (null? lst) (null? (cdr lst)))
      lst
      (let ((mid (quotient (length lst) 2)))
        (merge (sort (take mid lst) less?)
               (sort (drop mid lst) less?)))))

(sort (list 3 1 4 1 5 9 2 6) <)  ;; → (1 1 2 3 4 5 6 9)
```

Technical Architecture

Codebase Statistics

Component	Lines of Code	Purpose
LLVM Codegen	29,352	IR generation, compilation
Parser	3,293	Tokenizer, AST construction
Arena Memory	934	Memory management
REPL JIT	630	Interactive execution
Standard Library	~300	Higher-order utilities
Math Library	~450	Linear algebra, numerical methods
Total	**~35,000**	Complete implementation

Compilation Pipeline



Type System

Eshkol uses a **polymorphic tagged value system** at runtime:

- **13 value types**: NULL, INT64, DOUBLE, CONS_PTR, DUAL_NUMBER, AD_NODE_PTR, TENSOR_PTR, LAMBDA_SEXPR, STRING_PTR, CHAR, VECTOR_PTR, SYMBOL, CLOSURE_PTR
- **Exactness tracking**: Scheme-compatible exact/inexact number distinction

- **Mixed-type lists**: Heterogeneous data fully supported
- **Homoiconicity**: Lambda S-expressions preserved for metaprogramming

Why Eshkol?

For Machine Learning Researchers

- **First-class autodiff**: No library imports, no tape management
- **Vector calculus operators**: Built-in gradient, jacobian, hessian, divergence, curl, laplacian
- **Native performance**: LLVM compilation, no interpreter overhead

For Scientific Computing

- **Numerical precision**: Scheme's exact/inexact number semantics
- **Matrix operations**: Comprehensive linear algebra support
- **Math library**: Integration, root finding, eigenvalues

For Real-Time Systems

- **No GC pauses**: Arena allocation with deterministic cleanup
- **Predictable latency**: No stop-the-world garbage collection
- **Efficient memory**: 16-byte aligned tagged values

For Functional Programming Enthusiasts

- **Scheme heritage**: Familiar S-expression syntax
- **First-class functions**: Closures with proper lexical scoping
- **Homoiconicity**: Code as data, metaprogramming ready
- **Interactive development**: Full-featured REPL with JIT

For Systems Programmers

- **C FFI**: Direct foreign function calls
- **LLVM backend**: Native code generation
- **Library mode**: Compile as shared libraries
- **Low-level control**: Arena memory, no hidden allocations

Comparison with Alternatives

Feature	Eshkol	PyTorch	JAX	Julia	TensorFlow
Syntax	Scheme	Python	Python	Julia	Python
Compilation	AOT/JIT	JIT	JIT	JIT	Graph
Autodiff	Built-in	Library	Library	Library	Library
Vector Calculus	Built-in	Manual	Manual	Manual	Manual
GC	None	Yes	Yes	Yes	Yes
Homoiconicity	Yes	No	No	No	No
Closures	Native	Limited	Limited	Native	No

Roadmap

- [x] Core language implementation
- [x] LLVM code generation
- [x] Automatic differentiation (3 modes)
- [x] Vector calculus operators
- [x] Interactive REPL with JIT
- [x] Standard library
- [x] Math library
- [ ] GPU acceleration (CUDA/Metal)
- [ ] Distributed computing
- [ ] IDE integration (LSP)
- [ ] Package manager

Getting Involved

- **Repository**: https://github.com/tsotchke/eshkol

- **Issues**: Report bugs, request features
- **Pull Requests**: Contributions welcome

License

Eshkol: Where functional programming meets scientific computing.