# EasyLang Programming Language: Comprehensive Reference Manual

Soumyapriya Goswami
Department of Information Technology
Kalyani Government Engineering College

Version 1.0
October 15, 2025

## License



## Biography

**Soumyapriya Goswami** is a researcher and technologist pursuing his **B.Tech in Information Technology** at **Kalyani Government Engineering College (KGEC)**. His work spans **Artificial Intelligence**, **Quantum Computing**, and **IIoT**, focusing on intelligent, secure, and energy-efficient systems.

He is the author of *NASH-DQNSleep: Reinforcement Learning-based Quantum-Aware Sleep Scheduling Framework for Industrial IoT*, published in the **Elsevier Q1 journal – Engineering Applications of Artificial Intelligence (EAAI)**, which integrates **quantum-inspired reinforcement learning** and **energy-harvesting dynamics** to enhance industrial IoT efficiency.

Soumyapriya's open-source contributions, including **EasyLang**, a minimalist interpreted language, reflect his commitment to accessible computing education. Prior experience in embedded systems and scripting automation shaped EasyLang's emphasis on simplicity and performance.

He combines **AI-driven optimization**, **quantum resilience**, and **sustainable computing principles** to design adaptive systems for resource-constrained environments, with experiments in **Google Colab**, **Python frameworks**, and **Qiskit**. His vision is to pioneer **AI–Quantum hybrid frameworks** advancing intelligence, security, and sustainability in IoT, healthcare, and aerospace, contributing to **6G and beyond**.

**Email:** soumyapriya.goswami.it2023@kgec.ac.in
**LinkedIn:** linkedin.com/in/soumyapriyagoswami

# Contents

# 1 Introduction

## 1.1 Overview of EasyLang

EasyLang represents a paradigm of minimalist programming language design, prioritizing syntactic elegance, semantic clarity, and computational efficiency for instructional and prototyping applications. As an interpreted language, it eschews compilation overhead, enabling immediate execution via a bespoke runtime environment implemented in ANSI C. The core innovation lies in its recursive descent parser, which facilitates deterministic syntax analysis with minimal lookahead, ensuring robust error detection and recovery cues.

This manual provides an exhaustive exposition of EasyLang's constructs, semantics, and implementation intricacies, catering to practitioners seeking not only surface-level usage but also deeper architectural insights. By abstracting away ceremonial syntax prevalent in more verbose languages, EasyLang empowers users to articulate algorithmic intent with precision and brevity.

## 1.2 Core Design Principles

The language adheres to the following tenets:

- **Syntactic Minimalism**: Absence of mandatory delimiters (e.g., semicolons) and indentation enforcement; statements delineate via natural terminators (newlines or periods), mitigating common novice errors.

- **Semantic Transparency**: Dynamic typing with implicit coercion, where operations adapt contextually (e.g., polymorphic `+` for arithmetic or concatenation), fostering intuitive expression without explicit type annotations.

- **Interpretive Efficiency**: A single-pass evaluation over an abstract syntax tree (AST) derived from predictive parsing, optimizing for low-latency execution in resource-constrained environments.

- **Pedagogical Accessibility**: Constructs mirror natural language patterns (e.g., `if condition then action else alternative end`), accelerating comprehension for novices while retaining expressiveness for experts.

- **Implementation Modularity**: Lexer, parser, and evaluator compartmentalized for extensibility; the corpus ( 500 LOC) exemplifies concise, maintainable systems programming.

## 1.3 Audience and Prerequisites

This reference suits:

- Novice programmers transitioning from pseudocode.

- Educators curating curricula on foundational concepts.

- Systems developers prototyping domain-specific languages (DSLs).

Familiarity with basic algorithms and C-level constructs enhances appreciation of the interpreter's internals.

### 1.4 Invocation and Environment

To instantiate the runtime:

```
gcc -std=c99 -O2 -lm -o easylang easylang.c
./easylang <source.elang>
```

The interpreter ingests the source, tokenizes via finite state automata, parses into an AST via recursive procedures, and evaluates via depth-first traversal. Non-interactive by design, it interfaces stdin/stdout for I/O.

## 2 Lexical Analysis: Tokenization Fundamentals

Lexical analysis, the inaugural phase of compilation, transmutes source text into a sequence of tokens, abstracting away whitespace and annotations. EasyLang's lexer employs a deterministic finite automaton (DFA) for linear scanning, achieving O(n) complexity where n denotes input length.

### 2.1 Token Taxonomy

Tokens classify as:

- **Identifiers**: Alphanumeric sequences commencing with a letter or underscore (`[a-zA-Z_][a-zA-Z0-9_]*`). Canonicalized to lowercase; reserved keywords preempt identifier interpretation.
- **Literals**:
  - Numerics: Signed decimals ( `[`$\pm]?[0-9]+(\dot{[}0-9]*)?|\dot{[}0-9]+), parsedas$`double`.
  - `Strings: Doubly-quoted enclosures ("[`$^{"}$<br>$]|$<br>$."*"), supportingescapes($`\n`, `\"`).
- **Keywords**: Immutable lexicon (`set`, `if`, `while`, et al.), disambiguated post-scanning.
- **Operators**: Dyadic/monadic symbols (`+`, `==`, `-`), lexed as single characters or digraphs (`<=`).
- **Punctuators**: Structural markers (`( ) .`), with newlines as implicit terminators.
- **Annotations**: `#`-initiated comments, excised during scanning.
- **EOF**: Sentinel for termination.

### 2.2 Scanning Algorithm

The lexer advances via a position index, peeking the current character:

**Algorithm 1** Lexical Scanning Pseudocode

```
procedure LexNextToken
    while isWhitespace or Comment do
        advance position
    end while
    if EOF then return T_EOF
    end if
    if isAlpha then return lexIdentifier isDigit return lexNumber " return lexString
    elsereturn lexOperator
    end if
end procedure
```

Comments (`#` to newline) and whitespace (`isspace()`) are elided. Line numbers track for diagnostics.

### 2.3 Disambiguation and Edge Cases

- Keyword vs. Identifier: Longest match post-lowercasing; `Set` → `T_SET`. - Numeric Ambiguity: `123.` → valid (123.0); `.` → invalid. - Escapes: `\t` → tab, preserving fidelity.

This phase yields a token stream, primed for syntactic scrutiny.

## 3 Syntax Analysis: Parsing and AST Construction

Syntax analysis erects an abstract syntax tree (AST) from tokens, validating conformance to the language grammar. EasyLang deploys a recursive descent parser, a top-down strategy wherein parsing functions mirror production rules, invoking subparsers recursively. This approach, LL(1)-compliant (left-to-right, leftmost derivation with 1-token lookahead), obviates backtracking, yielding linear-time parsing.

### 3.1 Grammar Formalism

The grammar, articulated in Extended Backus-Naur Form (EBNF), is left-factorized and unambiguous:

program ::= { statement }
statement ::= assignment | print_stmt | read_stmt |
if_stmt | while_stmt | expression terminator
assignment ::= “set″ identifier “to″ expression terminator
print_stmt ::= “print″ expression terminator
read_stmt ::= “read″ identifier terminator
if_stmt ::= “if″ compare “then″ block [ “else″ block] “end″ terminator
while_stmt ::= “while″ compare “do″ block “end″ terminator
block ::= { statement }
expression ::= term { addop term }
addop ::= “+″ | “−″
term ::= factor { mulop factor }
mulop ::= “∗″ | “/″ | “%″
factor ::= number | string | identifier |
“(″ expression “)″ | “−″ factor
compare ::= expression { compop expression } [ “and″ compare]
compop ::= “==″ | “! =″ | “<″ | “<=″ | “>″ | “>=″
identifier ::= letter { letter | digit | “_″ }
number ::= [ “−″] ( digits [ “.″ digits] | “.″ digits )
string ::= “”″ { char − “”″ − “*n*″ | “*n*″ any } “”″
terminator ::= newline | “.″
letter ::= [*a*-*zA*-*Z*] | “_″
digits ::= [0-9]+
char ::= printable ASCII

This context-free grammar (CFG) is regular for lexing, LL(1) for parsing—FIRST/FOLLOW sets ensure unique predictions.

## 3.2 Parsing Mechanics: Recursive Descent in Depth

Each non-terminal spawns a dedicated procedure, consuming tokens via `advance()` and verifying via `expect()`. Lookahead via `peek_token()` resolves alternatives.

**Pseudocode for Key Parsers:**

**Algorithm 2** If-Statement Parser

```
procedure parseIf
    expect(T_IF)
    cond ← parseCompare()
    expect(T_THEN)
    body ← parseBlock()
    elseBody ← null
    if peek == ID("else") then
        advance()
        elseBody ← parseBlock()
    end if
    expect(T_END)
    return IfNode(cond, body, elseBody)
end procedure
```

For expressions, Pratt parsing could augment, but descent suffices: `parseExpression` chains left-recursive additives post `parseTerm`.

**Error Resilience:** Mismatched tokens trigger diagnostics with line/token context; no recovery—halt on first infraction for purity.

### 3.3 AST Representation

Nodes form a typed hierarchy:

- Statements: SetNode(var, expr), PrintNode(expr), IfNode(cond, body, else), etc.
- Expressions: BinaryNode(left, op, right), VarNode(name), NumNode(val), StrNode(val).

Serialization via pointers; evaluation recurses post-order.

This pipeline—lex → parse → AST—ensures syntactic fidelity.

## 4 Semantic Analysis and Evaluation: Bringing Code to Life

Post-parsing, semantics assign meaning: type resolution, coercion, execution.

### 4.1 Type System: Implicit and Polymorphic

Dynamically inferred: Numbers for ops, strings for literals. Coercion rules: - +: Num+Num → Num; Any+Str → Str. - Comparisons: Str → 0 (false); Num only otherwise.

No static checks—runtime harmony.

### 4.2 Evaluation Strategy: Tree Traversal

Depth-first, post-order: Leaves (literals/vars) compute first, propagating upward.

**Algorithm 3** Binary Expression Evaluation

```
procedure evalBinary(node)
    l ← eval(node.left)
    r ← eval(node.right) node.op + return coerceAdd(l, r) * return l.num * r.num    ▷
Num only == return (l.num == r.num) ? 1 : 0 . . .
end procedure
```

Variables resolve via linear search in a linked list—O(n) globals.

### 4.3 Control Flow Semantics

- If: Eval cond; branch on !=0. - While: Loop until cond=0; no tail optimization.

## 5 Advanced Topics: Grammar Derivations and Parsing Examples

This section delves into the theoretical underpinnings of EasyLang's syntax analysis, elucidating how the grammar derivations unfold during parsing and providing concrete examples of token-to-AST transformations. Understanding these mechanics not only illuminates the language's predictability but also equips advanced users with the knowledge to extend or debug the interpreter.

### 5.1 Derivation Trace: A Sample Parse

Consider the assignment statement `set x to 2 + 3 * 4.`. We trace its processing through the grammar productions, highlighting key decisions and AST construction.

1. **Lexical Phase (Tokenization)**: The lexer scans the input, yielding the token stream:
   - T_SET (keyword "set")
   - T_IDENTIFIER ("x", lowercased to "x")
   - T_TO (keyword "to")
   - T_NUMBER ("2", parsed as double 2.0)
   - T_PLUS (operator "+")
   - T_NUMBER ("3", double 3.0)
   - T_MUL (operator "*")
   - T_NUMBER ("4", double 4.0)
   - T_DOT (punctuator ".", serving as terminator)

   Whitespace and potential comments are elided; line tracking notes position 1.

2. **Syntactic Phase (Parsing Derivation)**: The parser matches the `assignment` production:

$$\begin{aligned} \text{assignment} &\Rightarrow \text{“set}'' \text{ identifier “to}'' \text{ expression terminator} \\ &\Rightarrow \text{“set}'' \text{ “}x'' \text{ “to}'' \text{ expression “.}'' \end{aligned}$$

   Now, derive the `expression`:

$$\begin{aligned} \text{expression} &\Rightarrow \text{term } \{ \text{ addop term } \} \\ &\Rightarrow \text{term “}+'' \text{ term} \\ \text{term (left)} &\Rightarrow \text{factor} \Rightarrow \text{number} \Rightarrow \text{“2}'' \\ \text{term (right)} &\Rightarrow \text{factor } \{ \text{ mulop factor } \} \\ &\Rightarrow \text{“3}'' \text{ “}*'' \text{ “4}'' \end{aligned}$$

   Precedence enforces `*` before `+`: The right `term` parses as a binary multiplication node before addition.

3. **AST Construction**: The parser erects a hierarchical node graph:

- Root: `SetNode` (var: "x", value: subexpression)
- Subexpression: `BinaryNode` (op: `+`, left: `NumNode(2.0)`, right: `BinaryNode` (op: `*`, left: `NumNode(3.0)`, right: `NumNode(4.0)`))

This tree encapsulates the derivation, ready for evaluation (yields 14.0).

### 5.2 Extended Derivation: Nested Conditional

For a more intricate construct, trace `if x > 5 then print "High" else set y to 10 end.`:

1. **Token Stream**: T_IF, T_IDENTIFIER("x"), T_GT, T_NUMBER("5"), T_THEN, T_PRINT, T_STRING("High"), T_IDENTIFIER("else"), T_SET, T_IDENTIFIER("y"), T_TO, T_NUMBER("10"), T_END, T_DOT.

2. **Production Derivation**:

$$\begin{aligned}
\text{if_stmt} &\Rightarrow \text{“if}'' \text{ compare “then}'' \text{ block “else}'' \text{ block “end}'' \text{ terminator} \\
\text{compare} &\Rightarrow \text{expression compop expression} \\
&\Rightarrow \text{“}x'' \text{ “} >'' \text{ “5}'' \\
\text{then block} &\Rightarrow \{ \text{ print_stmt } \} \Rightarrow \text{“print}'' \text{ “}High'' \\
\text{else block} &\Rightarrow \{ \text{ assignment } \} \Rightarrow \text{“set}'' \text{ “}y'' \text{ “to}'' \text{ “10}''
\end{aligned}$$

   The parser's lookahead distinguishes `else` from subsequent statements, invoking `parseBlock` recursively for each branch.

3. **AST Hierarchy**:

   - Root: `IfNode` (cond: `BinaryNode`(¿, Var("x"), Num(5.0)), body: `PrintNode`(Str("High")), else: `SetNode`("y", Num(10.0)))

   Evaluation: Cond yields 1/0; branch accordingly, executing leaf nodes (print or set).

These traces exemplify the grammar's predictive nature: At each step, the next token uniquely selects the production, averting ambiguity.

### 5.3 Parsing Diagnostics and Recovery Strategies

Upon mismatch (e.g., missing `end`), the parser emits a localized error: "Parse error: expected 'end' but found token T_IDENTIFIER('foo') at line 7". No speculative recovery—termination preserves integrity, though future iterations could synchronize via longest-prefix matching.

## 6 Feature Use Cases: Real-World Applications

EasyLang's parsimony belies its versatility across domains. Below, we delineate use cases, underscoring how its features synergize for pragmatic solutions.

### 6.1 Educational Tooling: Algorithmic Pedagogy

In classrooms, EasyLang demystifies iteration and conditionals. Instructors prototype exercises like bubble sort:

```
set n to 5  # Array simulation via globals
set arr_1 to 64; set arr_2 to 34; ...  # Manual array
set i to 1
while i < n do
    set j to 1
    while j < n - i do
        if arr_j > arr_{j+1} then
            # Swap logic
            set temp to arr_j
            set arr_j to arr_{j+1}
            set arr_{j+1} to temp
        end
        set j to j + 1
    end
    set i to i + 1
end
print "Sorted!"  # Visualize via prints
```

The absence of array syntax encourages explicit loops, reinforcing fundamentals.

### 6.2 Prototyping Scientific Computations

For ad-hoc simulations (e.g., population growth), EasyLang's arithmetic and loops suffice:

```
read initial_pop
read rate
read years
set pop to initial_pop
set y to 0
while y < years do
    set pop to pop * (1 + rate)
    print "Year " + y + ": " + pop
    set y to y + 1
end
```

Precision via doubles supports modest models; extend with

### 6.3 Embedded Scripting: Configuration and Automation

In IoT or scripts, EasyLang automates tasks like sensor polling:

```
set threshold to 75
set temp to 0  # Simulated read
while temp < threshold do
    read temp  # From sensor
    if temp > 80 then
        print "Alert: Overheat!"
    else
        print "Normal: " + temp
    end
    # Delay loop implicit
end
```

Lightweight footprint ( 500KB binary) suits resource-poor devices.

### 6.4 Data Processing Pipelines

Batch processing CSV-like data via manual loops:

```
set sum to 0
set count to 0
read value
```

```
while value != -1 do  # Sentinel end
    set sum to sum + value
    set count to count + 1
    read value
end
if count > 0 then
    print "Average: " + (sum / count)
end
```

Concat for reports; scale with globals as “arrays.”

## 6.5 Game Logic Prototypes

Simple text adventures:

```
set health to 100
set room to 1
while health > 0 do
    print "Room " + room + ". Health: " + health
    read action
    if action == "fight" then
        set health to health - 20
    else if action == "heal" then
        set health to health + 30
    else
        print "Unknown command."
    end
    set room to room + 1
end
print "Game over!"
```

Loops and ifs drive state; read for interactivity.

## 6.6 Financial Modeling

Monte Carlo simulations:

```
read trials
read initial
set total to 0
set t to 0
while t < trials do
    set current to initial
    set step to 0
    while step < 12 do  # Months
        set change to (rand() - 0.5) * 2  # Simulated rand [-1,1]
        set current to current * (1 + change / 100)
        set step to step + 1
    end
    set total to total + current
    set t to t + 1
end
print "Avg return: " + (total / trials)
```

(Note: `rand` hypothetical; use loops for pseudo-random.)

## 6.7 Text Analysis Tools

Word counters:

```
read text  # Assume string read extension
set count to 0
set pos to 1
```

```
while pos <= length(text) do  # Hypothetical len
    if is_space(text[pos]) then
        set count to count + 1
    end
    set pos to pos + 1
end
print "Words: " + count
```

Concat for reports; future strings enable full NLP lite.

## 6.8 Configuration Parsers

INI-like readers:

```
set key to ""
set value to 0
read line
while line != "END" do
    if starts_with(line, "[") then
        print "Section: " + trim(line)
    else
        # Parse key=value
        set eq_pos to find(line, "=")
        if eq_pos > 0 then
            set key to substr(line, 1, eq_pos - 1)
            set value to substr(line, eq_pos + 1)
            print key + "=" + value
        end
    end
    read line
end
```

Loops process lines; globals store configs.

## 6.9 Statistical Aggregators

Mean/variance:

```
read n
set mean to 0
set sum_sq to 0
set i to 0
while i < n do
    read x
    set mean to mean + x
    set sum_sq to sum_sq + x * x
    set i to i + 1
end
set mean to mean / n
set var to (sum_sq / n) - (mean * mean)
print "Variance: " + var
```

Arithmetic chains compute stats.

## 6.10 Embedded DSLs for Reports

Generate CSV:

```
set rows to 3
set r to 0
while r < rows do
    set val1 to r * 2
    set val2 to r * 3
```

```
    print val1 + "," + val2  # Concat for output
    set r to r + 1
end
```

Prints: 0,02,34,6

These use cases leverage EasyLang's core for diverse, low-overhead applications.

# 7 Sample Programs

Herein reside ten canonical programs, exemplifying idiomatic usage. Each includes commentary and expected output.

## 7.1 1. Hello World with Personalization

```
# Greets user by name
read name
print "Greetings, " + name + "! Welcome to EasyLang."
```

**Input:** Alice **Output:** Greetings, Alice! Welcome to EasyLang.

## 7.2 2. Arithmetic Calculator

```
# Basic four-function calc
read a
read op
read b
if op == "+" then
    print a + b
else if op == "-" then
    print a - b
else if op == "*" then
    print a * b
else if op == "/" then
    if b != 0 then
        print a / b
    else
        print "Error: Division by zero"
    end
else
    print "Unknown operator"
end
```

**Input:** 10 + 5 **Output:** 15

## 7.3 3. Factorial Computation

```
# Iterative factorial
read n
set fact to 1
set i to 1
while i <= n do
    set fact to fact * i
    set i to i + 1
end
print "Factorial of " + n + " is " + fact
```

**Input:** 5 **Output:** Factorial of 5 is 120

### 7.4 4. Prime Number Checker

```
# Trial division primality test
read num
set is_prime to 1
if num <= 1 then
    set is_prime to 0
else
    set i to 2
    while i * i <= num and is_prime == 1 do
        if num % i == 0 then
            set is_prime to 0
        end
        set i to i + 1
    end
end
print num + (is_prime == 1 ? " is prime" : " is not prime")
```

(Note: Ternary hypothetical; use if for output.) **Input:** 17 **Output:** 17 is prime

### 7.5 5. Fibonacci Sequence

```
# First n Fibonacci numbers
read n
set a to 0
set b to 1
set i to 0
while i < n do
    print a
    set temp to a + b
    set a to b
    set b to temp
    set i to i + 1
end
```

**Input:** 8 **Output:** 011235813

### 7.6 6. Simple Bank Account

```
# Balance manager (excerpt)
set balance to 0
read action
while action != "quit" do
    if action == "deposit" then
        read amt
        set balance to balance + amt
    else if action == "withdraw" then
        read amt
        if amt <= balance then
            set balance to balance - amt
        end
    else if action == "balance" then
        print balance
    end
    read action
end
```

**Input Sequence:** deposit 10050**Output:** 10050

### 7.7 7. Greatest Common Divisor

```
# Euclidean algorithm
read a
read b
while b != 0 do
    set temp to b
    set b to a % b
    set a to temp
end
print "GCD: " + a
```

**Input:** 48 18 **Output:** GCD: 6

### 7.8 8. Palindrome Checker

```
# String reverse check (hypothetical reverse func)
read s
set rev to ""  # Simulate reverse
set i to length(s) - 1
while i >= 0 do
    set rev to rev + substr(s, i, 1)
    set i to i - 1
end
if s == rev then
    print "Palindrome!"
else
    print "Not a palindrome."
end
```

(Extend for full strings.) **Input:** radar **Output:** Palindrome!

### 7.9 9. Linear Search

```
# Search in 'array' globals
set arr_size to 5
set arr_1 to 3; set arr_2 to 7; ...  # Fixed
read target
set found to 0
set pos to 1
while pos <= arr_size and found == 0 do
    if arr_pos == target then
        set found to 1
    end
    set pos to pos + 1
end
print (found == 1 ? "Found at " + (pos - 1) : "Not found")
```

**Input:** 7 **Output:** Found at 2

### 7.10 10. Temperature Converter

```
# C to F
read celsius
set fahrenheit to (celsius * 9 / 5) + 32
print celsius + " C  = " + fahrenheit + " F "
```

**Input:** 0 **Output:** 0°C = 32°F

These exemplars span paradigms, from computation to interaction.

# 8 Limitations and Future Work

## 8.1 Current Limitations

EasyLang's austerity imposes constraints:

- **Global Scope Exclusivity**: No lexical scoping; variable shadowing absent, risking namespace pollution in extended scripts.
- **Absence of Aggregate Types**: Lacks arrays, maps; simulations via indexed globals inefficient for scale.
- **String Subsetting Deficit**: Concatenation robust, but no indexing/slicing/length—text processing rudimentary.
- **Runtime Diagnostics Sparse**: Fatal on errors; no exceptions or partial execution.
- **Precision and Range**: Double semantics cap integer exactness at $\approx 2^{53}$; no arbitrary-precision.
- **I/O Constraints**: Stdin/stdout only; file ops, networking precluded.
- **Concurrency Void**: Sequential execution; no coroutines or parallelism.

## 8.2 Future Work: Evolutionary Roadmap

Prospective enhancements:

- **Modular Scoping**: Introduce functions with local bindings, enabling `def foo(x)  return x * 2; .`
- **Collection Primitives**: Arrays (`arr[3] = 5`) and maps for data structures.
- **Enhanced Diagnostics**: Try-catch, warnings, and interactive debugging.
- **Extended Numerics**: BigInt integration via GMP for cryptography.
- **I/O Augmentation**: File read/write, HTTP client for web tasks.
- **REPL Mode**: Interactive shell for exploratory coding.
- **Standard Library**: Modules for math (trigonometry), strings (regex), and utilities.
- **Performance Optimizations**: JIT compilation or VM for loops.

These trajectories preserve minimalism while amplifying utility.