Build Your Own LISP: Understanding How Programming Languages Work

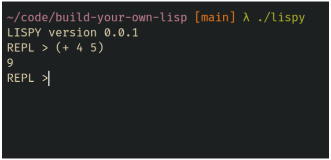
Amir Mohammad Taati

November 13, 2025

Demo

Here in this picture you can see our demo. You can see:

- A working REPL
- Reading input
- Parsing input
- Evaluating input
- Printing results



Who am I?

- My name is Amir Mohammad
- I'm a first year electrical engineering student
- I'm passionate about computer science

Motivation: What is This Talk About?

Our goals in this talk are:

- Get to know LISP
- Understand what a programming language is
- Understand different parts of a programming language
- Actually build a language!

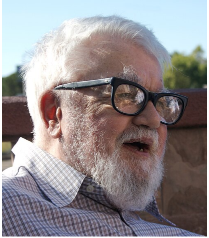
Why LISP?

- LISP stands for list processing
- It is simple
- Perfect for understanding how languages work

It's a Workshop

Tell me and I forget, teach me and I may remember, involve me and I learn

A Brief History of LISP



- Created by John McCarthy in 1958 at MIT
- Based on Alonzo Church's lambda calculus
- Second-oldest high-level programming language (after Fortran)

Example Code

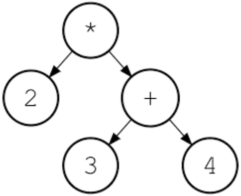
```
(+ (* 2 3) (- 4 1))
; => 9
```

Notice: Everything is in parentheses!

What is an S-Expression?

- Stands for Symbolic Expression
- The fundamental structure of Lisp code and data
- Everything in Lisp is an S-Expression

Example




```
(* 2 (+ 3 4))
; => 14
```

Structure

Two forms:

1. Atoms — indivisible values
   - Examples: `42`, `x`, `t`
2. Lists — collections of atoms or other lists
   - Lists are enclosed in parentheses
   - Format: `(operator operand1 operand2 ...)`

The Idea of Recursion

A process that defines itself through itself

Factorial: A Self-Referential Definition

$$n! = \begin{cases} 1, & \text{if } n = 0 \\ n \times (n-1)!, & \text{otherwise} \end{cases}$$

Notice how factorial is defined using itself!

In Lisp

```
(defun factorial (n)
  (if (= n 0)
      1
      (* n (factorial (- n 1)))))
```

This is recursion in action!

Grammar as Generation

Grammar is powerful:

- It's a set of rules that can generate infinitely many sentences
- Recursion allows those rules to refer to themselves
- It doesn't just describe; it produces

The MU Puzzle

Can you get from MI to MU?
You start with: `MI`
Your goal: produce `MU`
You have 4 rules. Can you do it?

The MIU System: Rules

Rule 1: If you have `xI`, you can add `U` at the end

`xI → xIU`

Rule 2: If you have `Mx`, you can double what comes after M

`Mx → Mxx`

Rule 3: Replace `III` with `U`

`xIIIy → xUy`

Rule 4: Remove `UU`

`xUUy → xy`

Generating with MIU Rules

Starting from `MI`:

```
Generation 1:  MI
               ↓ (apply Rule 2: double after M)
Generation 2:  MII
               ↓ (apply Rule 2 again)
Generation 3:  MIIII
               ↓ (apply Rule 3: III → U)
Generation 4:  MUI
               ↓ (apply Rule 1: add U)
Generation 5:  MUIU
```

Notice: Rules apply to their own output — recursion!

The MU Puzzle: Spoiler!

Can you reach MU from MI?

The MU Puzzle: Spoiler!

Can you reach MU from MI?

No! It's impossible.
Why? All strings keep a number of I's divisible by powers of 2.

- MI has 1 I
- Rules only multiply or reduce I's by 3
- MU has 0 I's — unreachable!

From MIU to Programming Languages

The same principle works for code:

- MIU rules generate valid strings
- Grammar rules generate valid programs
- Both use recursion
- Both start from simple rules

Now let's see LISP's grammar...

Grammar as Rules

A grammar consists of:

- Symbols — the building blocks
- Rules — how symbols can be replaced or combined
- A start symbol — where generation begins

LISP Grammar

```
<expression> ::= <atom> | <list>
<atom>       ::= <number> | <symbol>
<list>       ::= '(' <expression>* ')'
<number>     ::= [0-9]+
<symbol>     ::= [a-zA-Z+\-*/]+
```

Notice: `<expression>` appears inside `<list>` — recursion!

Example: Generating Valid LISP

Starting from `<expression>`:

1. Choose `<list>` rule
2. Get `'(' <expression>* ')'`
3. Each `<expression>` can be an atom or another list

This generates: `(+ 1 (* 2 3))`

What is Parsing?

The main goal of parsing:
Turn text into a data structure we can work with

`"(+ 1 2)"` → `['+', 1, 2]`

The Big Idea

Parsing $\rightarrow$ Turning code into a tree

- Code is text
- The tree is structure
- Structure gives meaning
- You can traverse a tree

Tokenizing

First step: Break into tokens

A token is the smallest meaningful unit of code

Example:

- Input string: `"(+ 1 (* 2 3))"`
- Tokens: `['(', '+', '1', '(', '*', '2', '3', ')', ')']`

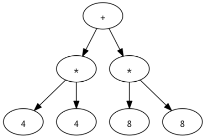
Abstract Syntax Tree (AST)

What is an AST?

- A tree data structure
- Represents the structure of the code
- Shows relationships between parts

Example: `['+', 1, ['*', 2, 3]]`

Example of AST



How We Parse

The algorithm:

- When we see (, we know a list is starting
- Read tokens until we see)
- If we see another (, recursively parse that list
- Return the complete list

Pipeline: S-Expressions $\rightarrow$ Tokens $\rightarrow$ AST

Recursive Descent

This is the parsing technique we use

Key ideas:

- Each grammar rule becomes a function
- Functions call each other following the grammar structure
- Recursion handles nested structures naturally

Our Grammar (Review)

```
<expression> ::= <atom> | <list>
<list> ::= '(' <expression>* ')'
<atom> ::= <number> | <symbol>
```

Grammar → Functions

Direct mapping:

- `<expression>` → `parse_expression()`
- `<list>` → `parse_list()`
- `<atom>` → `parse_atom()`

Each grammar rule becomes a function!

What is Evaluation?

Taking the AST and computing the result

`['+', 1, 2]` $\rightarrow 3$

Core Idea

The evaluation process:

1. Evaluate the expression
2. Apply the operator to the operands
3. Repeat recursively

The Evaluator Function

Two cases:

- If it is an atom $\rightarrow$ return the atom
- If it is a list $\rightarrow$ evaluate the list
    - First item is the operator
    - Rest are operands
    - Recursively evaluate each operand

Let's Practice

Trace through this expression:

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

Let's Practice

Trace through this expression:

`(+ 1 (* 2 3))`

Steps:

1. Evaluate `1` → `1`
2. Evaluate `(* 2 3)` → `6`
3. Apply `+` to `[1, 6]` → `7`

Overall Structure

What we'll build:

1. Tokenizing — String → Tokens
2. Parsing — Tokens → AST
3. Evaluating — AST → Result

Let's start coding!

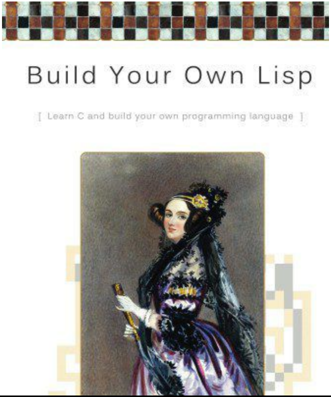
Workshop Flow

Today's agenda:

- Implement the tokenizer (already done!)
- Build the parser together
- Build the evaluator together
- Test our interpreter
- Celebrate!

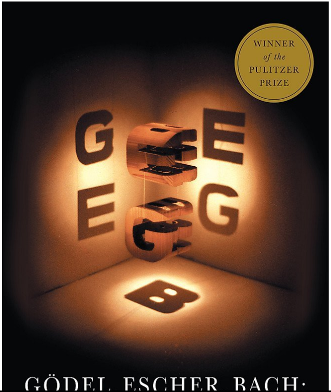
Build Your Own LISP

Author: Daniel Holden



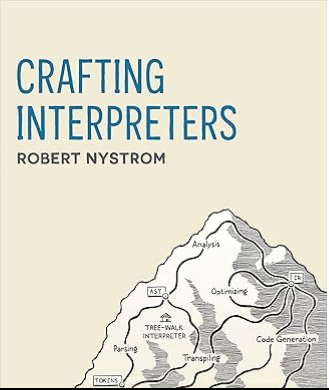
Gödel, Escher, Bach

Author: Douglas Hofstadter



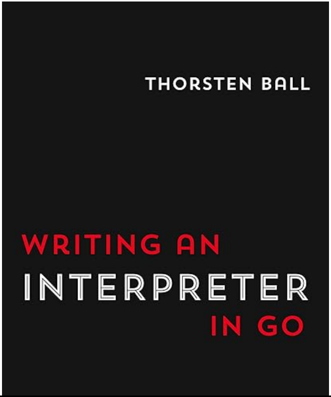
Crafting Interpreters

Author: Robert Nystrom



Writing an Interpreter in Go

Author: Thorsten Ball



What We've Built

Congratulations!

- A working LISP interpreter
- ~150 lines of Python
- Understanding of how languages work

Key Takeaways

The big ideas:

- Recursion is everywhere (grammar, parser, evaluator)
- Simple grammar = simple parser
- Code is just data with structure
- You can build a language!

Questions?

Thank you for attending!

Feel free to ask anything about:

- LISP
- Parsing
- Programming languages
- The implementation