Scripting Languages — Syntax, Semantics, Pragmatics, and Emacs Lisp

1. Introduction to Scripting Languages

When discussing programming languages, three major areas are essential:

Category	Description	Scripting Example	Traditional Example
Syntax	Structure or form of code (e.g., semicolons, braces).	Python: whitespace, Shell: no braces	C++: braces, semicolons
Semantics	Meaning or behavior of the program—what the code does.	Python: dynamic typing, Shell: command execution	C++: static typing, compiled execution
Pragmatics	Practical concerns: efficiency, configuration, usability, security, and interoperability. Most emphasized in scripting.	Python: glue code, Shell: automation	C++: performance, static analysis

- Scripting languages focus heavily on pragmatics, not just syntax or semantics.
- Pragmatics includes: how to get things working, configuration, efficiency, security, trust, and integration with other modules.

Pragmatics in Action:

> Suppose you need to automate a nightly backup. In C++, you would need to write, compile, and deploy a program, handling all errors and edge cases. In Bash, a 3-line script using `rsync` and a cron job suffices. The scripting approach wins on pragmatics: speed, integration, and ease of change.

---

2. Overview of Example Scripting Languages

Language	Primary Use	Integration	Performance	Extensibility	Example Code
Shell	OS automation, scripting	Unix tools, files	Low	High	`echo Hello`
Emacs Lisp	Editor extension	Emacs internals	Low-Medium	Very High	`(message "Hi")`
Python	General scripting, glue	C/C++ modules, APIs	Medium	High	`print("Hi")`
JavaScript	Web, extension, general	Browsers, Node.js	Medium-High	High	`console.log("Hi")`

Bytecode and JIT:

- **Bytecode:** An intermediate, platform-independent code executed by a virtual machine (e.g., Python `.pyc`, Java `.class`).
- **JIT (Just-In-Time) Compilation:** Converts bytecode to native machine code at runtime for speed (e.g., JavaScript V8, Java HotSpot).

```
[Source Code]
     |
     v
[Bytecode] --(JIT)--> [Native Machine Code]
```

---

3. Key Themes of Scripting Languages

Ease of Use

- Scripting languages prioritize accessibility and quick onboarding.
- Easy to learn, fast to write short scripts (e.g., "Hello World!").
- Low barrier to entry increases ecosystem participation.

Reliability

- Scripting languages avoid worst-case crashes common in C++ (e.g., null pointer exceptions, subscript errors).
- Instead, they raise exceptions or print errors.
- **Downside:** Permissiveness may allow incorrect programs to execute, decreasing logical reliability.

Language	Crashes on Subscript Error?	Throws Exception?
C++	Yes	No
Python	No	Yes
Shell	No	Typically alerts

Example:

- C++:

```
int arr[2] = {1,2};
int x = arr[5]; // Undefined behavior, may crash
```

- Python:

```
arr = [1,2]
x = arr[5] # Raises IndexError
```

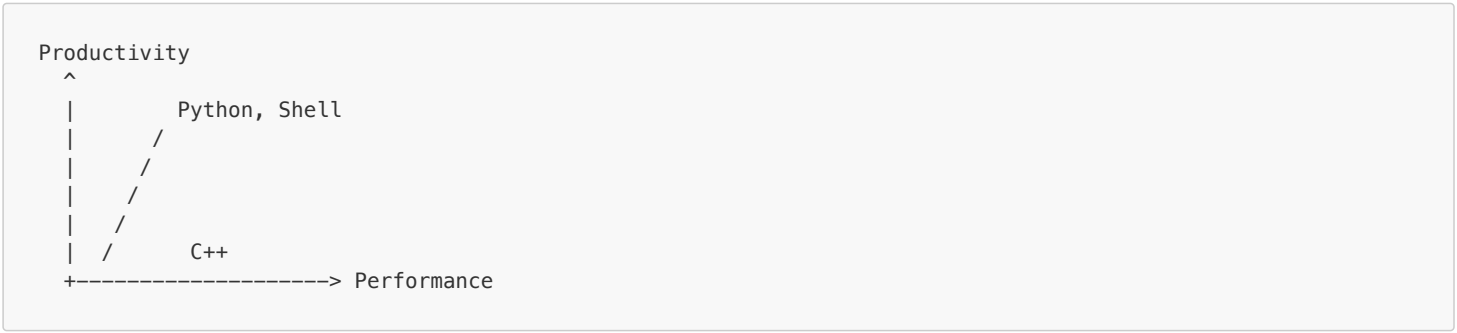
Scalability

- Good for small to medium programs, glue code, and lightweight data transformation.
- Problems with very large codebases due to lack of strict typing and fragmented tooling.
- Large systems often use scripting languages to configure or glue together performant code written in C++ or similar.

Performance

- Generally slower than compiled languages due to runtime checks and interpreted execution.
- Tradeoff: slower performance for higher productivity and reliability.
- JIT compilers and bytecode interpreters (e.g., JavaScript, Emacs Lisp) help mitigate the performance gap.
- Example: C++ is fast because it doesn't do subscript checking; Python checks at runtime and throws exceptions.

Productivity vs. Performance:



---

4. Concepts and Philosophical Differences

General vs. Extension Languages

Language Type	Example	Description	Example Code
General Purpose	Python, Java	Use it for almost anything	`print(42)`
Extension	JavaScript, ELisp	Embedded in host applications (e.g., browser, editor) to extend functionality	`(message "Hi")` or `alert('Hi')`

Extension Language Example:

- JavaScript in a browser:

```
document.body.style.background = 'yellow';
```

- Emacs Lisp in Emacs:

```
(set-background-color "yellow")
```

Traditional vs. Scripting

Feature	Scripting Languages	Traditional Languages
Performance	Low	High
Flexibility	High	Medium
Error Handling	Graceful	Harsh (e.g. segfaults)
Compilation	Optional or runtime only	Mandatory
Type System	Dynamic/optional	Static/mandatory
Tooling	Fragmented, lightweight	Mature, integrated
Deployment	Script file, interpreter	Compiled binary

Design Purpose of Languages

Language	Intended Use
JavaScript	Extending browser functionality
Shell	Automated OS-level task and config scripts
Emacs Lisp	Extending the editor's features
SQL	Query and data specification
Rust	High-performance and safe system programming

---

5. Emacs Lisp: An Introduction

Historical Context and Purpose

- Lisp was created for AI in the 1960s, focusing on self-modifying and program-generating code.
- Emacs Lisp is a variant tailored to the GNU Emacs text editor.
- Emacs Lisp is an **extension language**: enhances and customizes Emacs behavior.

Emacs Lisp as an Extension Language

- Deep integration with Emacs internals: buffers, windows, syntax highlighting, etc.
- Provides primitives focused on editing rather than general computing (compare to SAP's payroll-specific extension language).
- Emacs is built on:
  - Low-Level C libraries
  - Lisp Interpreter written in C

Emacs Architecture:

```
+--------------------+
|   Emacs (C Core)   |
+--------------------+
         |
         v
+--------------------+
| Lisp Interpreter   |
+--------------------+
         |
         v
+--------------------+
| User Lisp Code     |
+--------------------+
```

- Code entered in Emacs is interpreted or compiled into bytecode.

Self-Modifying Code Example:

```
(defun add-x (n)
  (defun x () n))
(add-x 42)
(x) ; => 42
```

---

6. Syntax and Semantics in Lisp

Lisp Syntax Characteristics

- Simple, uniform syntax using parenthesized prefix notation.
- All function calls look like: `(function arg1 arg2 ...)`
- No operator precedence, no infix notation.
- Symbols and data use homogenous parenthetical syntax.

Structuring Code as Lists

- Code and data are blurred: both written as lists.
- Example:
  - Data: `'('a b c)`
  - Code: `(+ 34 25)`

Symbols and Special Forms

- **Symbols**: Unique named atomic values (e.g., `nil`, `t`, `my-var`).
- **Special Forms**: Control structures and definition constructs that are not function calls (e.g., `defun`, `let`, `lambda`, `quote`).

Special Form	Purpose	Example
defun	Define a function	`(defun foo (x) x)`
let	Local variable binding	`(let ((x 1)) x)`
lambda	Anonymous function	`(lambda (x) (+ x 1))`
quote	Prevent evaluation	`'(a b c)`
if	Conditional	`(if cond a b)`
progn	Sequence of expressions	`(progn a b c)`

Macro Example

```
(defmacro when (cond &rest body)
  `(if ,cond (progn ,@body)))

(when (> 3 2)
  (message "3 is greater than 2"))
```

Expressions vs. Data

- `'expression` prevents evaluation.
- `(eval expression)` forces evaluation even of quoted data.

Form	Meaning
'abc	Symbol `abc` as data
abc	Evaluate variable `abc`
'(a b c)	List of three symbols a, b, c
(quote (a b c))	Same as above

Interactive Evaluation (Scratch Buffer)

- Evaluated using:
  - `C-J`: Evaluate previous expression and show result below.
  - `C-x C-e`: Evaluate expression at point and show in mini-buffer.

Debugging Basics

- Emacs will drop into the debugger upon errors.
- Key command to exit debugger: `C-]` (Control + closing square bracket)
- Debugger sessions can nest; exit each with `C-]` until you return to the top level.

---

7. Emacs Lisp Data Structures

Numbers

- Integers: Arbitrary precision (bignums)
- Floats: IEEE 64-bit doubles
- Example: `(+ 10000000000000000 1)`
- Integer division truncates; floating-point division if one operand is a float.
- No support for smaller-width floats (e.g., 8-bit) as in some ML applications.

Strings

- Written with double quotes
- Supports newlines: `"Hello\nWorld"`
- Strings can contain any characters, including newlines.
- Functions like `format` allow C-style string formatting.

Symbols

- Named values that point to variables or functions
- Used for identifiers, constants, or stand-alone atoms
- Special symbols:
  - `nil`: false, end of list
  - `t`: true
- Each symbol is unique by name; two symbols with the same name are the same object.

Lists

- Fundamental concept in Lisp: "List Processing"
- Built from pairs (cons cells): each has a value and a pointer to the next cell.
- Built-in functions for lists:
  - `cons`: Construct a new pair
  - `car`: Return first element
  - `cdr`: Return rest of list

List Example

```
(cons 'a (cons 'b nil)) ; => (a b)
```

Improper Lists

- If the last element is not `nil`, it's an improper list.

```
(cons 'a 'b) ; => (a . b)
```

- `(a . b)` is the printed form for an improper list.

Improper List Diagram:

```
+-------+-------+
|  a    |   o---|---> b
+-------+-------+
```

Quoting and Data/Code Duality

- `'abc` yields the symbol `abc` as data.
- `'('a b c)` yields the list `(a b c)` as data.
- `(eval ...)` can turn data into code.

Every Paren Counts

- In Lisp, every parenthesis is meaningful. Extra parentheses change the meaning (e.g., `(n)` is a function call to `n`, not just the value of `n`).

Vectors

- Fixed-size, indexable arrays

- Created via: `(make-vector 5 'x)`
- Less commonly used than lists, but allow random access.

Hash Tables, Character Tables, Markers, Buffers, Frames, Windows, Processes

- **Hash Tables**: Key-value associative maps.
- **Character Tables**: Hash tables with character keys for efficiency.
- **Markers**: Pointers to specific positions in buffers that adapt as buffer changes.
- **Buffers**: Editable text containers (contents of a file, scratch, etc.).
- **Frames/Windows**: Emacs GUI elements.
- **Processes**: Represent running subprocesses (e.g., shells) under Emacs.
- **Fonts, X widgets, Terminals**: Other Emacs object types.

Example: Creating and Using a Marker

```
(setq m (make-marker))
```

- `m` is now a marker object. To make it point to a buffer, pass a buffer argument.

Example: Buffers

```
(current-buffer) ; returns the current buffer object
```

Comparison Table: Lists, Vectors, Hash Tables

Structure	Ordered?	Random Access	Typical Use	Example Creation
List	Yes	No	Sequence, stack, queue	`(list 1 2 3)`
Vector	Yes	Yes	Fixed-size, fast lookup	`(make-vector 3 'x)`
Hash Table	No	Yes (by key)	Key-value mapping	`(make-hash-table)`

---

8. Functional Programming Concepts in Emacs Lisp

Functions

- First-class objects
- Created via `lambda` or `defun`

Using `defun`:

```
(defun add-three (x) (+ x 3))
```

Interactive Commands:

- Use `interactive` keyword to make a function callable via UI/keystrokes.

```
(defun greet-user ()
  (interactive)
  (message "Hello!"))
```

Example:

```
(defun show-buffer-name ()
  (interactive)
  (message (buffer-name)))
```

Local Bindings (`let`)

- Creates local variables bound to evaluated values.

```
(let ((x 3) (y 4)) (+ x y)) ; => 7
```

- Temporarily masks global variables if names overlap.
- `let` can bind multiple variables; each binding is a pair `(name value)`.
- If only a symbol is given, it is bound to `nil`.

let vs. Global Variable Table:

Feature	let (local)	Global Variable
Scope	Expression/block	Everywhere
Lifetime	Temporary	Until unset
Side Effects	None (isolated)	Can affect all code
Example	`(let ((x 1)) x)`	`(setq x 1)`

Lambda Expressions and Closures

- Emacs supports anonymous functions with `lambda`.

```
(lambda (x y) (+ x (* y y)))
```

- Can pass lambdas as arguments or assign to variables.
- **Closures:** Functions that capture their lexical environment.

```
(let ((n 10))
  (lambda (x) (+ x n))) ; Returns a function that adds 10
```

Special Forms

- Not all parenthesized expressions are function calls. Some are special forms (e.g., `lambda`, `let`, `quote`, `defun`).
- Special forms have unique evaluation rules.

---

9. Emacs Lisp Utilities and Commands

Emacs Commands

- Functions callable via UI must be declared interactive:

```
(defun my-command ()
  (interactive)
  (message "Hello"))
```

- Can query keystrokes or input with `interactive` argument.

Accessing Documentation

- `C-h k`: Describe key bindings.
- `C-h f`: Describe function.
- `C-h m`: Describe current mode.
- Mouse over source file link (e.g., window.el) to open implementation in Emacs.

Useful Emacs Lisp Introspection Commands:

Command	Description
`describe-function`	Show documentation for a function
`describe-variable`	Show documentation for a variable
`apropos`	Search for functions/variables by name
`find-function`	Jump to function source code

Command	Description
`eldoc-mode`	Show function signatures in minibuffer

10. Summary

This lecture introduced scripting languages, emphasizing their focus on pragmatics—real-world concerns of integration, performance trade-offs, ease of use, reliability, and extensibility. It differentiated scripting languages from general-purpose and low-level languages, exemplified by Emacs Lisp and Python. The lecture gave an in-depth presentation of Emacs Lisp as an extension language, explaining its evaluation model, key data types (including symbols, lists, strings, functions), and its support for self-modifying code, which allows it to script and modify Emacs in powerful ways. The class also demonstrated fundamental Lisp programming techniques including quoting, evaluation, function definition, lambda expressions, local variable bindings, and debugging tools within Emacs.

Visualizations and Diagrams

List Structure in Lisp (ASCII Diagram)

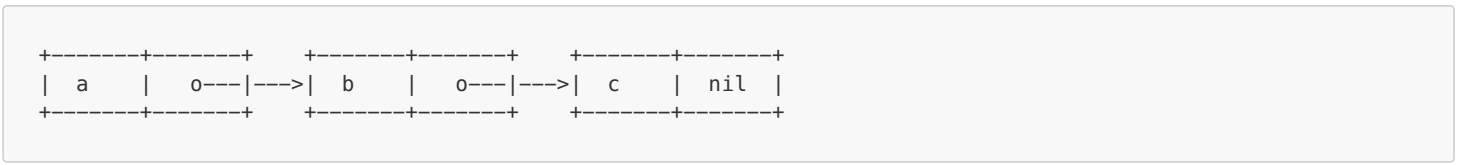


Table: Lisp Data Types

Type	Example	Notes
Integer	42, 100000000000000000	Arbitrary precision
Float	3.14, -0.5	IEEE 64-bit double
String	"Hello\nWorld"	Can contain newlines
Symbol	'abc, 'nil, 't	Unique by name
List	'(a b c), (cons 'a nil)	Linked list of cons cells
Vector	(make-vector 5 'x)	Fixed-size, indexable array
Hash Table	(make-hash-table)	Key-value map
Marker	(make-marker)	Buffer position pointer
Buffer	(current-buffer)	Editable text container
Frame/Window	(selected-frame)	GUI elements
Process	(start-process ...)	Running subprocess

Table: Special Forms vs. Function Calls

Syntax Example	Is Function Call?	Notes
(+ 1 2)	Yes	Calls function +
(defun foo ...)	No (special form)	Defines a function
(let ((x 1)) ...)	No (special form)	Local variable binding
(lambda (x) (+ x 1))	No (special form)	Anonymous function
(quote (a b c))	No (special form)	Returns data, not code

Table: Emacs Lisp Evaluation Commands

Command	Effect
C-J	Evaluate previous expression, show result below
C-x C-e	Evaluate expression at point, show in mini-buffer
C-]	Exit debugger
C-h k	Describe key bindings

Command	Effect
C-h f	Describe function
C-h m	Describe current mode

Table: Comparison of Scripting and Traditional Languages

Feature	Scripting Languages	Traditional Languages
Performance	Low	High
Flexibility	High	Medium
Error Handling	Graceful	Harsh (e.g. segfaults)
Compilation	Optional or runtime only	Mandatory

Language Types Summary Diagram

```
[General Purpose]
      |
      v
[Extension Language] <---[Host Application]
      |
      v
[Scripting Language]
```

Macro Expansion and Evaluation Process

```
[Source Code]
     |
     v
[Macro Expansion]
     |
     v
[Evaluation]
     |
     v
[Result]
```