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

1. Introduction to Scripting Languages

Definitions and Context

- Scripting languages are often thought of as languages used to automate processes, configure, or extend large systems.
- Traditional classifications of programming languages include:

Category	Description
Syntax	Refers to the structure or form of code (like semicolons, braces in C++). Considered solved and relatively easy.
Semantics	Refers to the meaning or behavior of the program—what the code does.
Pragmatics	Practical concerns of programming - efficiency, configuration, usability, security, and interoperability. Most emphasized in scripting.

Scripting languages emphasize pragmatics more than traditional programming languages.

---

2. Overview of Example Scripting Languages

Shell

- One of the oldest scripting languages, designed to automate and configure Unix programs.
- Still widely used to run and link other programs.

Emacs Lisp

- Not originally a scripting language, but adapted as one.
- Originated as a general purpose language designed to write AI programs in the 1960s.
- Adopted for scripting Emacs due to its support for program generation and self-modifying code.
- Known as an extension language, designed specifically for Emacs, not general-purpose use.

Java

- Not a scripting language but contributed vital implementation ideas:
  - Bytecode abstraction
  - Interpreters and Just-In-Time (JIT) compilation
- Strong typing, performance-oriented, reliable
- Emphasizes reliability and strict compiled checking of code.

JavaScript

- Originally designed to script and extend web browsers.
- Became a language with general-purpose capabilities.
- Uses similar implementation techniques as Java (e.g., JIT, bytecode)

Python

- Now one of the world’s most popular languages (estimated 25% of developer mindshare).
- Designed as a general-purpose scripting language.
- Emphasizes code readability, quick prototyping, and integration of external modules
- Supports linking external libraries (often written in C/C++)

---

3. Key Themes of Scripting Languages

Ease of Use

- Scripting languages prioritize accessibility.
- Easy to learn, fast to write short scripts.
- Example: Python makes it easy for beginners to write “Hello World!” without extensive setup.

Reliability

- Scripting languages often avoid worst-case crashes common in languages like C++ (e.g., null pointer exceptions).
- Instead, they raise exceptions or print errors.
- However, permissiveness may allow incorrect programs to execute without failure, decreasing logical reliability.

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

Scalability

- Scripting languages are good for:
  - Small to medium programs
  - Glue code
  - Lightweight data transformation
- Problems occur with very large codebases due to:
  - Lack of strict typing
  - Fragmented tooling
- Integration often includes low-level languages (like C++) for performance and reliability in large systems.

Performance

- Generally slower than compiled languages due to runtime checks and interpreted execution.
- Tradeoff: slower performance for higher productivity
- JIT compilers and bytecode interpreters are used to mitigate this gap (e.g., JavaScript, Emacs now includes bytecode compilation).

---

4. Concepts and Philosophical Differences

General vs. Extension Languages

Language Type	Example	Description
General Purpose	Python, Java	Use it for almost anything
Extension	JavaScript, ELisp	Embedded in host applications (e.g., browser, editor) to extend functionality

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

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, created for AI in the 1960s, grew to support 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

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

---

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 blurred: both written as lists.
- Example:
  - Data: ’(a b c)
  - Code: (+ 34 25)

Symbols and Special Forms

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

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)

---

6. Emacs Lisp Data Structures

Numbers

- Integers: Arbitrary precision (bignums)
- Floats: IEEE 64-bit doubles
- Example: (+ 100000000000000000 1)

Strings

- Written with double quotes
- Supports newlines: “Hello\nWorld”

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

Lists

- Fundamental concept in Lisp: “List Processing”
- 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

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

Vectors

- Fixed-size, indexable arrays
- Created via: (make-vector 5 ’x)

Hash Tables, Markers, Buffers

- Hash Tables: Key-value associative maps.
- Markers: Pointers to specific positions in buffers that adapt as buffer changes.
- Buffers: Editable text containers (contents of a file, scratch, etc.)

---

8. Functional Programming Concepts

Functions

- First-class objects
- Created via lambda
- Using `defun`:

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

- Interactive Commands:
  - Use `interactive` keyword

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

- Example:

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

- `let`: Defines local variables

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

Lambda Expressions

- Emacs supports anonymous functions with `lambda`.

Example:

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

- Can pass lambdas as arguments or assign to variables.

Local Bindings (LET)

- Creates local variables bound to evaluated values.

Example:

```
(let ((x 5) (y 7))
  (+ x y))
```

- Temporarily masks global variables if names overlap.

---

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

---