Project 1 Lisp Interpreter

Sprint2: Extending the S-Expression

Goal

After completing the first sprint of defining the core data structure, the "Second Sprint" focuses on building the fundamental tools for working with S-expressions. The goal is to create essential constants and functions that allow you to identify, construct, and deconstruct these symbolic expressions. This is a critical step toward creating a functional interpreter or language.

📦 S-Expressions: The Core Data Types

S-expressions (sexps) are the fundamental data type in Lisp-like languages. They can be simple elements like **symbols**, **numbers**, or **strings**, or they can be complex nested structures called **lists**. The building block for a list is a **cons cell**, which holds two components: the `car` (the first element) and the `cdr` (the rest of the list). A special constant, `nil`, serves a dual purpose: it represents both the empty list and a boolean `false` value. A corresponding `truth` constant represents a boolean `true` value.

🛠️ Predicates: The Type Checkers

Predicates are simple functions that return a true or false value, allowing you to determine the type of an S-expression. This is essential for directing program flow and ensuring that functions operate on the correct data types.

- `nil?`: Checks if a sexp is the `nil` constant.
- `symbol?`: Checks if a sexp is a symbol.
- `number?`: Checks if a sexp is a number.
- `string?`: Checks if a sexp is a string.
- `list?`: A special case. A sexp is considered a list if it is either a **cons cell** or the `nil` constant, which represents the empty list.

In addition to these, a general `sexp_to_bool` function maps any S-expression to a boolean value. In this system, only `nil` is considered false; everything else is true.

🏗️ Constructors and Accessors: The Builders and Retrievers

These functions are the workhorses for creating and manipulating S-expressions.

- `sexp`: A versatile **constructor** that takes a string and automatically creates the correct type of sexp. For example, it can tell if a string represents a number (like "42"), a string (like `"hello"`), or a symbol (like "variable").
- `cons`: This constructor builds a **cons cell** by linking two S-expressions together. This is how you build a list one element at a time.
- `car`: An **accessor** that retrieves the first element from a list (or a cons cell).
- `cdr`: An **accessor** that retrieves the rest of the list from a list (or a cons cell).

☑ Proof of Concept

With these tools, you can perform basic operations to prove the system works. You can create different types of sexps and then build a complete list by nesting `cons` calls. You can then use the `car` and `cdr` accessors to retrieve the individual elements and the remainder of the list, confirming that the structure holds together as intended. This all sets the stage for the next sprint, where these primitives can be used to perform more complex operations.