本解释器实现了一种支持**渐进式类型（Gradual Typing）**的 Scheme 方言，融合了动态语言的灵活性与静态类型的安全性。该语言运行于自定义解释器之上，解释器使用 Racket 编写，并基于 S-expression 语法扩展。

---

1. 类型系统设计

类型系统以结构化表示方式实现，每种类型为一个透明结构体（`struct`）：

- 基本类型结构体：如 `type-int`、`type-bool`、`type-number`、`type-unknown`
- 函数类型：`type-function` 保存参数类型列表与返回类型
- 组合类型：支持 `type-list`、`type-pair` 与 `type-union`

核心操作包括：

- `type-equal?`：结构等价判断，允许 `type-unknown` 参与比较
- `type-compatible?`：实现宽松兼容性，用于支持渐进式类型检查，例如 `Int` 与 `Number` 可互相兼容

类型信息可通过 `(: expr type)` 或 `(define var : type val)` 注入，并在求值阶段进行动态验证。

---

2. 解释器结构

解释器基于经典的递归求值器构建，核心组件包括：

- **环境模型（Lexical Environment）**
  使用嵌套 `environment` 结构体链表示作用域，每个环境记录变量绑定（可能为 `box` 以实现递归定义的支持）。
- **表达式求值（`eval-expr`）**
  使用 `match` 分支匹配并处理各类表达式，支持 `lambda`、`define`、`if`、`let/let*`、`cond`、函数调用等。
- **闭包表示**
  函数以 `closure` 结构表示，记录参数、函数体与定义时环境。
- **类型包装**
  每个值可以被封装为 `typed-value`，其中携带运行时类型信息。动态检查通过 `infer-type`（推断值的类型）和 `type-compatible?` 完成。

---

3. Lambda 参数类型注入

解释器在遇到 `(define f : (T1 T2 -> ?) (lambda (x y) ...))` 时，会自动将 `T1`、`T2` 注入 `lambda` 的参数中，形成带显式类型的参数列表 `(lambda ((x : T1) (y : T2)) ...)`。

注入机制递归处理嵌套 lambda，确保嵌套函数中也可传递类型信息。

---

4. 函数返回类型推断

本解释器支持**受限的返回类型推断**，设计意图为增强类型信息，但**不对整个程序进行全局类型推导。**推断规则如下：

- 仅当函数类型注解为 `(-> ?)` 且函数体为表达式

- 参数类型必须是已注入的明确类型（来自 lambda 注解）

推断通过 `infer-expr-type` 实现，它分析表达式结构并根据操作数类型判断返回类型（如 `Int + Int -> Int`，否则返回 `Number`）。

在 `define` 处理中，若目标函数签名为 `(-> ?)` 且可推断出返回类型，则构造新的函数类型并更新绑定。

---

5. 模块化与运行支持

解释器主循环位于 `run-program`，支持：

- 从 `.rkt` 文件读取 S 表达式序列
- 顺序执行每个表达式，维护全局环境盒子
- 错误处理使用 `with-handlers` 包装，以捕获运行时错误并打印

提供对外接口：`run-program`、`run-file`、`eval-expr` 等，支持作为脚本运行或模块导入。