# 目录

1. [目录](#目录)
2. [引言](#引言)
3. [计算理论介绍](#计算理论介绍)
   1. [什么是计算？函数作为一种解释](#什么是计算函数作为一种解释)
   2. [lambda 演算：表示函数的最小逻辑系统](#lambda-演算表示函数的最小逻辑系统)
   3. [函数式编程：lambda 演算的图灵表述](#函数式编程lambda-演算的图灵表述)
4. [Lachesis 语言设计](#lachesis-语言设计)
   1. [S-表达式](#s-表达式)
   2. [Q-表达式](#q-表达式)
   3. [变量](#变量)
   4. [lambda 函数](#lambda-函数)
      1. [形式](#形式)
      2. [可变参数](#可变参数)
      3. [科里化](#科里化)
   5. [实用方法](#实用方法)
5. [解释器设计](#解释器设计)
   1. [工作模式](#工作模式)
   2. [系统框架](#系统框架)
   3. [运行流程](#运行流程)
6. [解释器实现细节](#解释器实现细节)
   1. [开发环境](#开发环境)
      1. [命令行软件](#命令行软件)
      2. [VIM](#vim)
      3. [make](#make)
      4. [vscode](#vscode)
   2. [容器设计](#容器设计)
   3. [语法分析器](#语法分析器)
   4. [建立函数抽象语法树](#建立函数抽象语法树)
   5. [推导抽象语法树推导](#推导抽象语法树推导)
   6. [函数的创建与执行](#函数的创建与执行)
      1. [创建预置函数](#创建预置函数)
      2. [全局变量和局部变量](#全局变量和局部变量)
      3. [创建lambda 函数](#创建lambda-函数)
      4. [执行函数](#执行函数)
      5. [开发新的预置函数](#开发新的预置函数)
   7. [错误处理](#错误处理)
   8. [动态调试器模式](#动态调试器模式)
   9. [标准库设计](#标准库设计)
      1. [从匿名函数到实名函数](#从匿名函数到实名函数)
   10. [补遗](#补遗)
   11. [帮助系统](#帮助系统)
7. [系统结果分析](#系统结果分析)
   1. [测试用例](#测试用例)
8. [结论](#结论)
   1. [不足与缺陷](#不足与缺陷)
      1. [神出鬼没内存错误](#神出鬼没内存错误)
      2. [O3优化编译报错](#o3优化编译报错)
      3. [缺少宽字符支持](#缺少宽字符支持)
      4. [import 文件路径逻辑](#import-文件路径逻辑)
      5. [宏](#宏)
9. [致谢](#致谢)
10. [参考文献](#参考文献)
11. [附录](#附录)

基于lambda 演算的函数式语言解释器的设计与实现

作者：高成志 &nbsp 指导老师：章林忠

（安徽农业大学 信息与计算科学专业 &nbsp 学号：18112722）

摘　要：

一个基于lambda 演算的函数式语言Lachesis 以及配套的解释器系统被设计出来。在解释器开发过程中，Lachesis 经历了几个阶段的变化，最终以表示一类符号表达式的lambda函数方案为基础，能够完成基本的求值、变量绑定、匿名函数等功能，并具有相当程度的可拓展性。在本文中，我们首先介绍了Lachesis 设计背后的计算理论和函数式语言的一般特点，然后描述了Lachesis 语言的关键特性和解释器设计的总体框架，最后展示了解释器实现中的一些技术细节。通过Lachesis 的开发，我们加深了对函数式编程的理解和……

关键词：

计算理论；计算机语言；解释器；UNIX 编程；

# 引言

如果你想从头做苹果派，你必须先创造宇宙——卡尔萨根（Carl Edward Sagan，1934年11月9日－1996年12月20日）【1】

~~定义一种语言的一种方法是编写标准规范文档，例如ECMAScrip，解释语言中允许的各种表达式以及它们是如何被推导的。这种技术的优点是，读者可以快速地吸收语言的一般概念，但通常很难从标准的描述中提取出语言的细节。~~

~~另一种方法是用某种元语言为它实现一个解释器。这种技术的优点是能够清楚完整地指定语言的细节，否则它就不可能被造出来。假设元语言有一个形式化的规范，那么解释器在该语言中就有一个形式化的含义，可以对其进行分析。~~

~~用于定义另一种语言的元语言不需要高效地执行，因为它的主要目的是向开发者解释另一种语言。元语言的基本数据构造也不需要用位和字节来定义。事实上，我们可以直接使用元语言的逻辑和集合理论来处理整个程序文本。~~

~~本文要实现一个函数式语言解释器，叫做Lachesis，取名自古希腊神话的命运三女神之一。~~

~~Lachesis 实现了一些函数式语言的最小功能集合，包括表达式求值，变量绑定，函数等。同时又有着极强的可拓展性，方便后来的开发者和用户添加新的功能和函数。~~

# 计算理论介绍

## 什么是计算？函数作为一种解释

首先，计算科学的抽象理论至少应该回答以下问题——如何处理“计算”的概念。对这个概念的解释有很多数学模型，基于有限状态机理论的图灵机可能是其中最重要的一个。从某种角度来看，这个模型是相当具体的，因为它建立了一个特定的计算机制，所以我们甚至可以观察到它在物理上是如何施行的。但是，图灵机模型限制了我们对计算的想象力，导致几乎所有的现代计算机都必须以相同的方式进行计算。

要克服这个限制，最好的方法是绕过计算的物理机制定义，仅仅根据输入和输出信息之间的关系来描述计算过程的概念。

关于计算应该满足的等价性的直觉告诉我们：如果一个问题可以通过一台计算机来解决，那么也就可以通过另一台计算机来解决。为了阐明这一点，我们把计算机的工作看作是计算函数，而计算就是一种调用可计算函数的过程。

## lambda 演算：表示函数的最小逻辑系统

为了表达函数的概念，数学家阿隆佐·邱奇（Alonzo Church）在 1930 年代发明了Lambda 演算，它是一个能用来表示函数本质的最小逻辑系统。

最基本的Lambda 演算系统包括

1. 构造 lambda 项（term）
2. 对lambda 项执行的归约操作（reduction operations）

Lambda 演算的**项**由以下三个规则定义：

| 语法 | 名称 | 描述 | 举例 |
| --- | --- | --- | --- |
| x | 变量 | 一个变量 x 本身就是一个 项。 | x、y、plus |
| ( x.M) | 函数抽象 | x为变量，M为函数体，也是一个 项。 | x.plus x x |
| MN | 函数应用 | 如果 M 和 N 都是 项，那么 MN也是一个 项。 | ( x .plus x x) y => (plus y y) |

或者用更简洁的巴科斯范式表示为：

重复应用上述三个规则就可以得到所有的 项。

Lambda 的**规约操作**则有两种：

| 操作 | 名称 | 描述 |
| --- | --- | --- |
| ( x.M[x]) -> ( y.M[y]) | 转换 | 重命名表达式中的绑定变量，用于防止命名冲突。 |
| (( x.M) E) -> (M[x:=E]) | 规约 | 用参数替换表达式中函数抽象（M）中的绑定变量。 |

如果能避免命名冲突（例如之后我们在解释器的实现中巧妙地做到的那样），那么 转换就不是必须的。而不断运用 规约所得到的最终结果将是一个 规范型，这种情况下将不能再进行规约操作。

Lambda 演算是图灵完备的，这意味着它有着和一般的计算机语言相当的表达能力，但却是用lambda 自身的编码形式。

光看lambda 演算的形式定义未免感到无所适从，我们来举一个例子演示lambda 演算到底是如何工作的。

布尔代数是一组元素（通常为两个，“TRUE”与“FALSE”）和服从定义的公理在这些元素上的运算（通常为与、或、非）。在lambda 演算中，“TRUE” 被编码为接受两个参数并返回第一个，而“FALSE”则是接受两个参数并返回第二个，具体来说就是：

$$TRUE \quad := \quad \lambda{x}.\lambda{y}.{x} \\ {FALSE \quad : = \quad \lambda{x}.\lambda{y}.{y}}$$

相对于C 语言中常用的宏定义（ TRUE=1 & FALSE=0 ）来说，这个对布尔型的定义抽象得令人无法理解，为什么“TRUE”是第一个？难道就不能是第二个吗？

答案是完全可以。事实上，这个定义恰好满足了布尔型的最小条件，即集合论断言子集和它的补集的交集为空集：

从这个角度上看，C 语言通过宏来定义布尔型的方式反而是不严谨的，因为{1}的补集不是{0}，而是除了1之外的所有整数的集合。

接下来我们可以编码布尔代数上的运算，例如非（NOT）定义为：

如果我们使用非进行运算，接收 TRUE 作为参数，计算过程如下 ：

$$ \begin{array}{lll}
\overbrace {NOT}^{函数}\quad \overbrace {TRUE}^{参数}& = \quad \overbrace{(\lambda{b}.b\enspace FALSE \enspace{TRUE})}^{函数} \enspace \overbrace{TRUE}^{参数} \\
& = \quad (TRUE) \enspace {FALSE}\enspace {TRUE} \\
& =\quad (\lambda{x}.\lambda{y}.{x})\enspace {FALSE} \enspace {TRUE} \\
& = \quad FASLE
\end{array} $$

其结果也是一个 规范型，且符合我们通常对布尔非运算的一般认识。

总而言之，lambda 演算给了我们机制和信心，能够只用函数计算解决其他指令式编程语言（基于图灵有限状态机模型）所能解决的事情。

## 函数式编程：lambda 演算的图灵表述

相对于我们所熟悉的指令式编程，函数式编程是一种完全不同的编程范式，它将计算视为函数的求值，从而避免了状态变更和使用可变数据。它的特点有：

* 函数是”一等公民”：与其他数据类型处于平等地位，可以赋值给其他变量，也可以作为参数传入另一个函数，或者作为别的函数的返回值。
* 只用“表达式”，不用“语句”：每一步都是单纯的运算，而且都有返回值。
* 没有“副作用”：函数要保持独立，所有功能就是返回一个新的值。

在函数式编程中，lambda演算（也就是匿名函数）被用作一个函数生成器：

let multipl = λxλy.x\*y   
let conbine = λxλy.xy  
;;; apply and eval  
multipl 2 3 -> 2 \* 3  
conbine ‘BYV’ ‘oid’ -> ‘BYVoid’

这为函数成为一等公民（和其他类型没有区别）提供了实现手段和理论依据。

考虑到大多数函数式语言要运行在依据图灵有限状态机模型设计出来的硬件上，这些硬件对基本的数据操作做了相当多的优化工作，因此在计算的最终阶段（例如2 \* 3）依然使用lambda 演算系统表达是十分不明智的。

$$
\begin{array}{lll}
2 \* 3 & = MULTIPLY \enspace 2\enspace3 \Rightarrow \overbrace{(λ abc.a(bc)) }^{MULTIPLY} \overbrace{(λ sz.s(s(z)))}^{2} \overbrace{(λ xy.x(x(x(y)))) }^{3} \\
& = λ c.(λ sz.s(s(z)))((λ xy.x(x(x(y))))c) \\
& = λ cz.((λ xy.x(x(x(y))))c)(((xy.x(x(x(y))))c)(z)) \\
& = λ cz.(λ y.c(c(c(y)))) (c(c(c(z)))) \\
& = λ cz.c(c(c(c(c(c(z)))))) \\
& = 6 \\
\end{array}
$$

$$
\begin{array}{lll}
2\*3 & = MUL \enspace 00010 \enspace 00011 \\
& = ……经过硬件加速的原码一位乘法 \\
& = 6
\end{array}
$$

因此，虽然函数编程语言最重要的基础是lambda 演算，但在具体实现上却也离不开图灵有限状态机。可以说，我们是在用指令式的工具表达函数式的思想。

函数式编程的另一个有争议性的来源是范畴论，这么说的原因是几大函数式语言（lisp、Haskell、Scala）在开发时并没有严格考虑过范畴论，但范畴论中的一些理论恰好能解释函数式编程的很多现象。

范畴论视角下的计算是范畴之间的转换，范畴则是包含了对象及对象之间关系的代数结构。在函数式语言的具体实现中，我们往往会用一个包含了数据和方法的容器来表达范畴的概念。

# Lachesis 语言设计

## S-表达式

表达式求值是程序语言要解决的基本问题之一，例如：

(\_input) >> 1 + 1  
(output) >> 2

事实上，鉴于绝大多数函数抽象语法树（我们的也不例外）采用的是波兰表达式的记号方法，为了方便推导，我们索性将Lachesis 也设计成波兰表示法的，也就是操作符置于操作数的前面的表示方法。

(\_input) >> + 1 1  
(output) >> 2

这样的好处是可以通过操作符赋值法轻松地实现可变参数个数的运算：

(\_input) >> + 1 1 1 1 1 1  
(output) >> 6

这里的 + 不再被看成一个二元运算符，而是一个表示加法函数的符号，也就是函数的名字。

这样的设计在涉及多运算符表达式时波兰表示法自身不需要加括号的优点就消失了，例如以下表达式不使用括号会产生歧义：

(\_input) >> \* + 1 1 3 2  
(output) >> error!  
(output) >> could be 10 ( [ \* [+ 1 1 3] 2 ] ) or 12 ( [ \* [ + 1 1 ] 3 2] )

所以Lachesis 的求值语句被设计成使用方括号作为界定符的符号表达式（S-表达式）

## Q-表达式

S-表达式一旦被读入就会开始求值，有时候我们并不想这样，例如表达式中含有未知变量时直接求解会抛出错误：

(\_input) >> + x 1   
(output) >> error! x is undefined!

而未知变量可能在之后定义，在这种情况下我们希望延迟表达式的推导而仅仅保留其形式（惰性求值），这时我们就需要 Q-表达式，用一对花括号定义：

(\_input) >> {+ x 1}  
(output) >> {+ x 1}

Q-表达式惰性求值的特性广泛运用在程序间传递信息，因为它不像S-表达式那样是转瞬即逝的。

## 变量

变量是一个很有误导性的名字，因为函数式语言中的变量实际上是不变的，一旦初始化后就不能更改。

Lachesis 中的变量概念更类似一个符号绑定，我们可以通过Q-表达式来传递变量的名字：

(\_input) >> def {x} 1   
(\_input) >> x  
(output) >> 1

我们设计的def 函数可以像golang 或Python那样一次给多个变量初始化：

(\_input) >> def {x y z} 1 2 3  
(\_input) >> print x y z  
(output) >> 1 2 3

在函数式语言中，函数也可以是变量，函数的形式用Q-表达式传递，需要时再用eval函数将Q-表达式转换成S-表达式并立刻推导。

(\_input) >> def {func\_name} {+ x 1}  
(\_input) >> def {x} 1  
(\_input) >> eval func\_name  
(output) >> 5

有了lambda 函数之后我们可以定义一些只在函数内部起作用的局部变量，以 =为标志符：

[func {max & xs} {  
 if [== [tail x] nil] {fst xs}  
 {do  
 [= {rest} [unpack max [tail xs]]]   
 [= {item} [first xs]]  
 [if [> item rest] [item] {rest}]  
} }]

rest、item 都是在函数内部生成的变量名，不会污染全局命名空间。

## lambda 函数

### 形式

将变量名绑定在一个表达式上的做法并不能很好地表达函数的概念，一个显而易见的缺点是会把变量名暴露参数到外部，lambda 函数则解决了这个问题。

(\_input) >> def {add\_two} [\ {x y} {+ x y}]  
(\_input) >> [ add\_two 1 2 ]  
(output) >> 3

lambda 函数以\ 为函数名，一个Q-表达式作为参数列表，一个Q -表达式为函数体。lambda 函数的变量在外部是不可见的，这有助于实现函数式语言纯函数的特性。

### 科里化

### 可变参数

## 实用方法

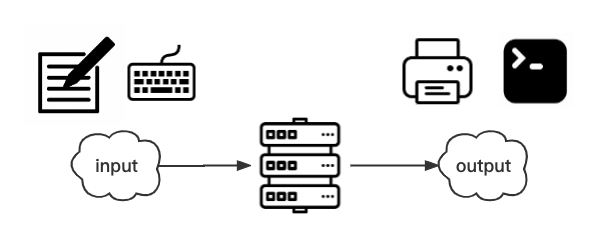
以上就是Lachesis 的核心语法概念，它简单得简直不像话，我们还需要定义一些程序内置的实用方法，列举如下：

| 接口 | 描述 | 举例 |
| --- | --- | --- |
| [+-\*/==!=><≥≤] <sexpr>\* | 常见数字运算符和比较符 | \* 1 5 => 5 |
| head <qexpr> => <qexpr> | 返回表达式第一个元素 | head {1 2 3} => {1} |
| tail <qexpr> => <qexpr> | 返回表达式除了第一个元素 | tail {1 2 3} => {2 3} |
| list <sexpr> => <qexpr> | 将S表达式转换成Q-表达式 | list [+ 1 2] => {+ 1 2} |
| join <qexpr> => <qexpr> | 将多个Q-表达式合并成一个 | join {+} {1 2} => {+ 1 2} |
| if <bool> <qexpr> <qexpr> => <sexpr> | 分支判断 | if [== x y] {+ x y} {- x y} |
| print <string>\* => <string> | 打印多个字符串 | print "hello world" => hello world |
| fprint <file\_name> <string>\* => <string> | 向文件打印多个字符串 | fprint "new.txt" "hello world" => new:hello world |
| import <string>\* | 导入其他lachesis文件 | import "stdlib" |

# 解释器设计

## 工作模式

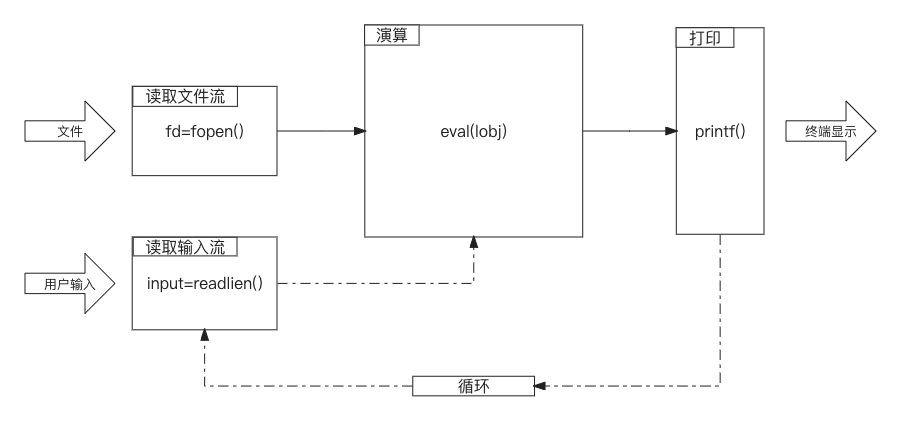
本系统支持现代解释器通常支持的两种工作模式，一个是交互式提示符（interactive prompt）式的读取-求值-输出循环（RELP：Read–eval–print loop），用户在命令行直接输入指令后马上返回结果。另一个则由用户事先写好脚本文件，解释器依次读取脚本中的内容并执行。



inter

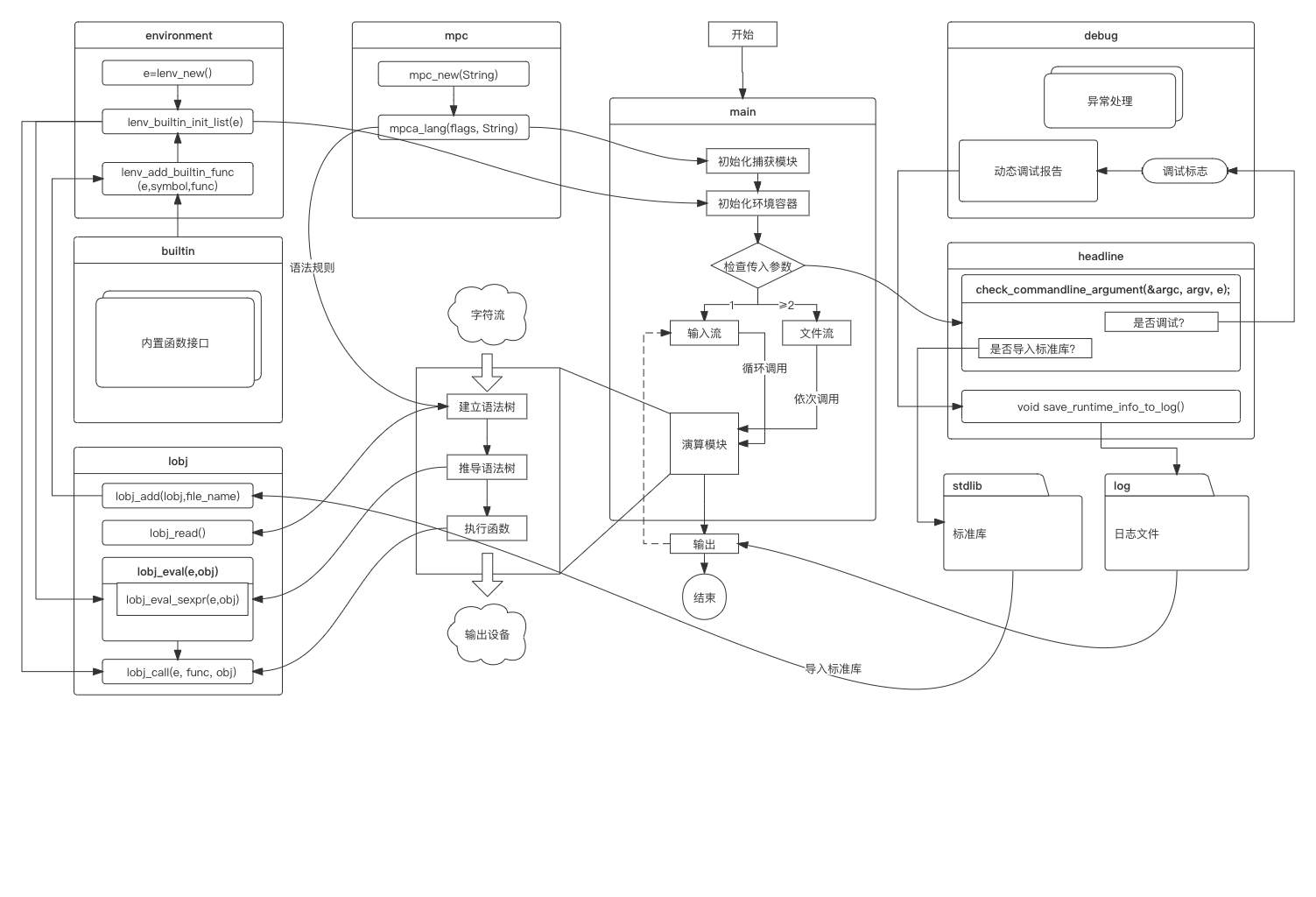
（两种工作模式）

在UNIX/Linux 设计哲学中，一切皆是文件，用户输入流不过是一个名为”stdin“的文件罢了。因此，除了在输入的捕获阶段要求开发者作不同的处理外，无论是普通文本类型的文件还是用户在终端的输入，都能使用同一套处理逻辑。



rad

## 系统框架

解释器的总体结构如 图上所示，整个解释器以main 函数为程序起点，检查输入的命令行参数后进入相应的工作模式，调用其它模块完成演算任务。图中各模块功能简要介绍如下 :

1. 环境模块（environment）：创建和初始化环境容器，添加符号和函数定义。
2. 内置函数模块（builtin）：系统保留字和预置函数的开发接口，方便未来功能拓展和二次开发。
3. 容器操作模块（lobj）：对Lachesis 对象的一系列初始化、转换、读取、求值、函数调用。
4. 语法解析器和词法分析器模块（mpc）：对输入字符流的捕获和抽象语法树的建立。
5. 调试模块（debug）：打印动态调试信息和用于处理语义异常的函数包。
6. 命令行交互模块（headline）：检查输入参数、导入标准库和帮助系统。
7. 标准库（stdlib）：用Lachesis 语言自身定义的一些实用函数。
8. 日志文件（log）：动态调试信息的默认输出区。

## 运行流程

# 解释器实现细节

## 开发环境

本系统在一台Darwin 内核的Inter MacBook Pro 上用C 语言开发开发，操作系统版本是Monterey 12.0.1。

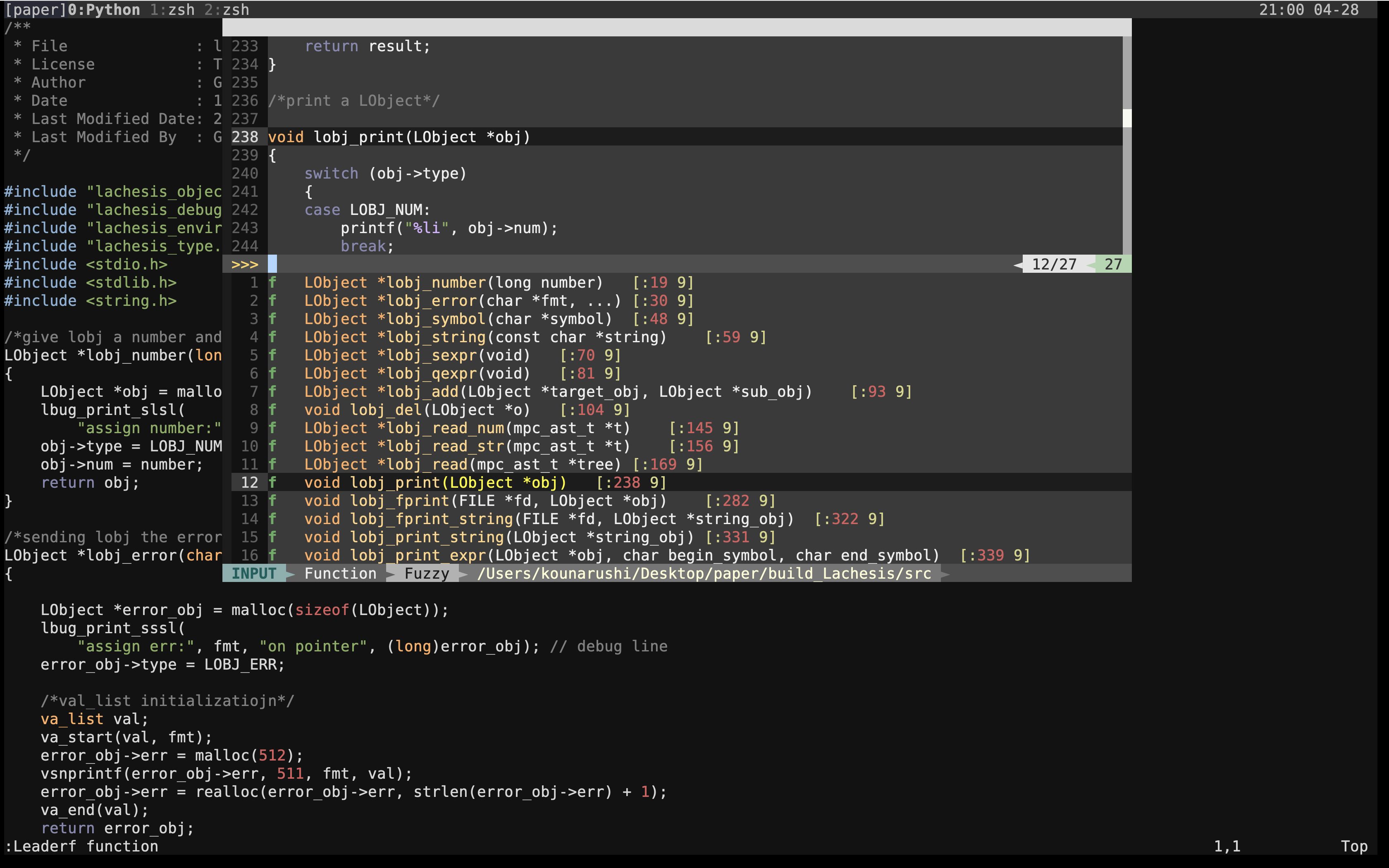
本系统所使用的一些开发技术介绍如下：

### 命令行软件

* vim：终端文本编辑器，本系统主要在vim 上开发完成。
* git：分布式版本控制系统，用于系统版本的备份、多路开发、回滚、同步。
* tmux：一个终端多路复用器，用于终端窗口管理、保存状态、远程协作等。
* ranger：一个带有vim键绑定的终端文件管理器，它提供了一个极简的TUI界面，带有目录层次结构视图。
* make： 编译工具，本系统主要将其作为一个指令宏使用。
* fzf：按名检索的高速路径跳查找器，它所提供的的通用接口可以用作各种作用，在本系统开发中通常用其进行高速路径跳转和打开文件。

### VIM

Vim 是一门上古神器，



Screen Shot 2022-04-28 at 9.00.29 PM

* leaderf：
* coc.nvim：

### make

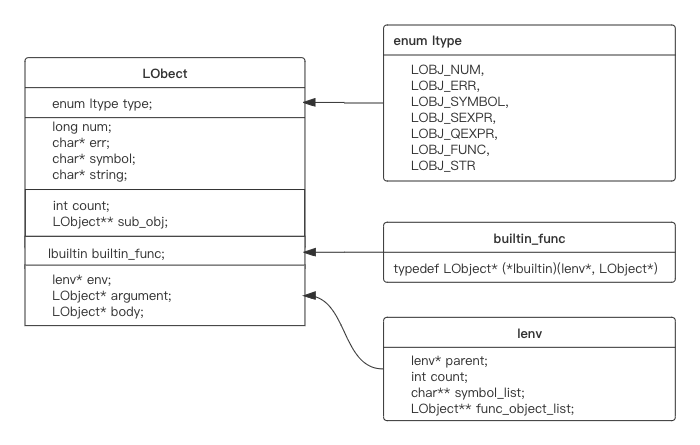
​

## 容器设计

Lachesis 解释器系统用一个名为LObject的结构体容器表达函数式编程中的“范畴”概念，执行计算的过程被抽象为范畴之间的转换。

LObject 容器中的内容列举如下：

1. 枚举类型enum ltype ：用来来识别该结构体的类型。
2. 基本数类型：容器携带的数据信息，依据枚举类型初始化赋值。
3. 指向其他LObject 容器地址的指针以及指针的个数。
4. 指向函数对象的函数指针。
5. 环境变量和存储函数对象信息。

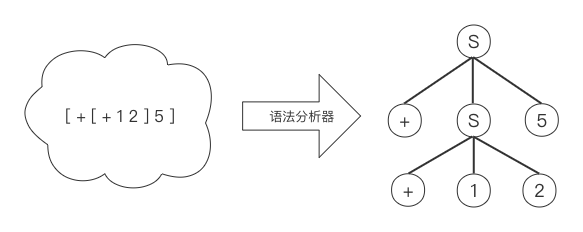


未命名文件 (2)

解释器内部传递的信息都由LObject包裹，

## 语法分析器

语法分析器的功能是将大量杂乱无章甚至含有非法输入的字符流通过我们自己指定捕获规则过滤后映射成函数抽象语法树。



未命名文件 (4)

我们没有自己开发专用的语法解析器，那将让整个工程和论文的内容无法控制地膨胀。一个第三方的语法解析库，mpc（Micro Parser Combinators）【mpc】，被用来执行Lachesis 的语法解析工作。

通过调用mpca\_lang() 函数，我们用正则表达式定义了九种语句的捕获规则，它们是Lachesis 所能识别的全部输入词符。

| 规则 | 定义 | 解释 | 举例 |
| --- | --- | --- | --- |
| number | /-?[0-9]+/ | 0或1个“-”开头，后跟任意个0-9之间的数字 | -1 |
| symbol | /[a-zA-Z0-9\_+\\-\*\\/\\\\=<>!&]+/ | 任意个字母、数字、\_+-\* /=<>!&的组合 | apple |
| string | /\"(\\\\. | [^\"])\*\"/ | 以\开头的单字符或除了”之外的任意字符 | “New” |
| comment | /;[^\\r\\n]\*/ | 以;开头的除了\r\n 之外的任意多个字符 | ; comment! |
| sexpr | '[' <expr>\* ']' | 被”[ ]“包裹起来的任意条<expr> | [ + 1 2 ] |
| qexpr | '{' <expr>\* '}' | 被”{ }“包裹起来的任意条<expr> | { + 1 2 } |
| expr | <number> | <symbol> | <string> | <comment> | <sexpr> | <qexpr> | 任意一条六种类型 | print |
| lexpression | /^/ <expr>\* /$/ | 任意多条expr | print “hello world” |

由上表可知，expr[ession] 只是前六种类型的囊括，而lexpression 是任意个expr[ession] 的列表。这样做的目的是方便在程序中统一捕获符合LExpression 规则的输入并绑定在一个指向函数抽象语法树的指针 raw 上。

（输入流中的捕获）

mpc\_result\_t raw;  
……  
if (mpc\_parse("<stdin>", input, LExpression, &raw)) {……}

（文件流中的捕获）

if (mpc\_parse\_contents(obj->sub\_obj[0]->string, LExpression, &raw)) {……}

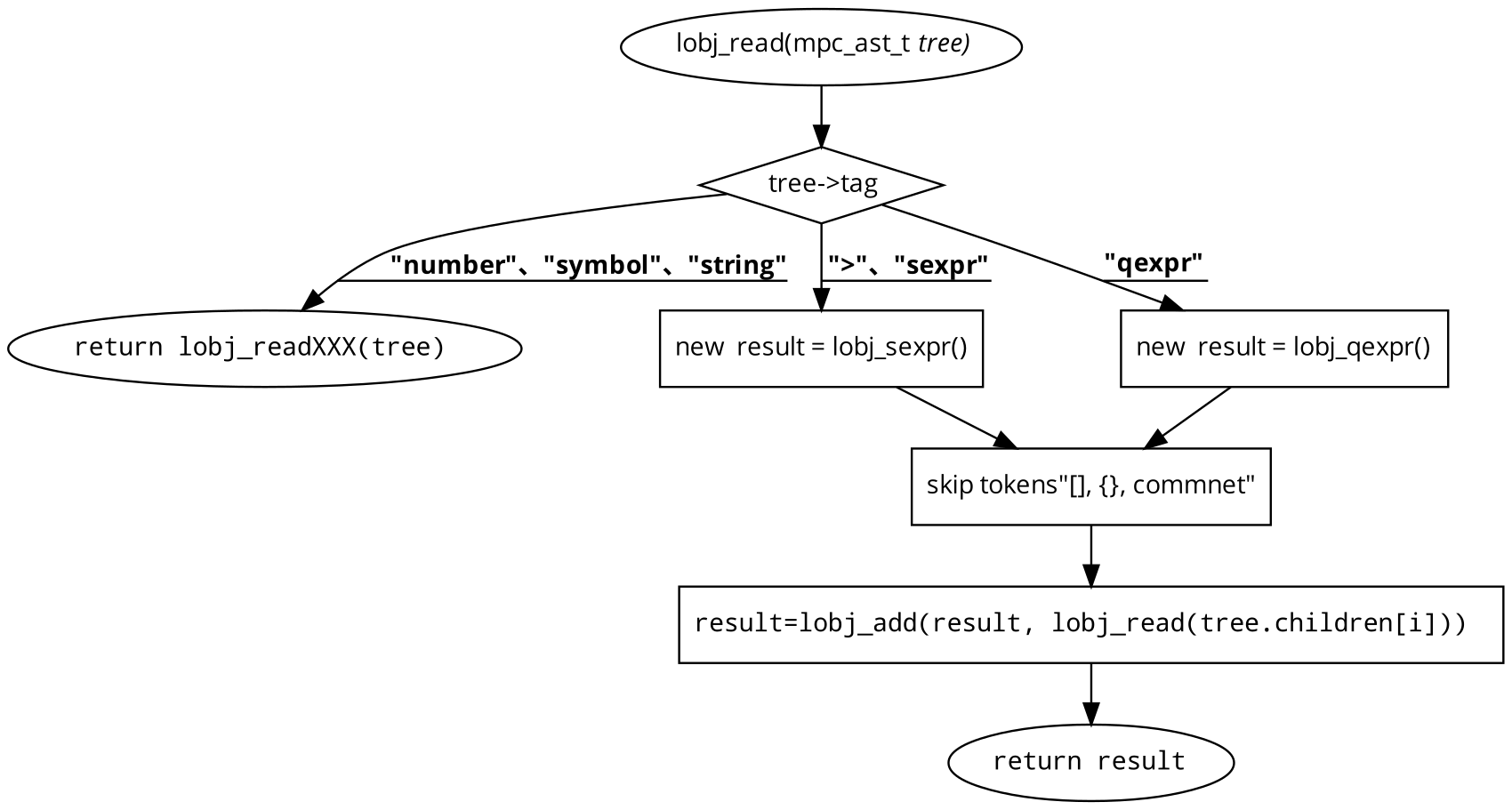
## 建立函数抽象语法树

语法分析器捕获的结果是一个多叉树，定义如下。其中tag 指示了捕获词符的类型，也就是我们所定义的八种捕获规则（number、symbol、string……）外加上 “>”、“regex” 等库定义词符。contents是具体捕获的内容，例如tag 为“number” 的结构体中contents 的内容是“13”。当tag 为“>”、“sexpr”或“qexpr”时，说明这是一个树根，需要递归读入其子树。

typedef struct mpc\_ast\_t {  
 char\* tag;  
 char\* contents;  
 mpc\_state\_t state;  
 int children\_num;  
 struct mpc\_ast\_t\*\* children;  
} mpc\_ast\_t;

函数lobj\_read() 被用来读取一棵抽象语法树的内容并转换成一棵Lobject 树。其算法和示意图如下：

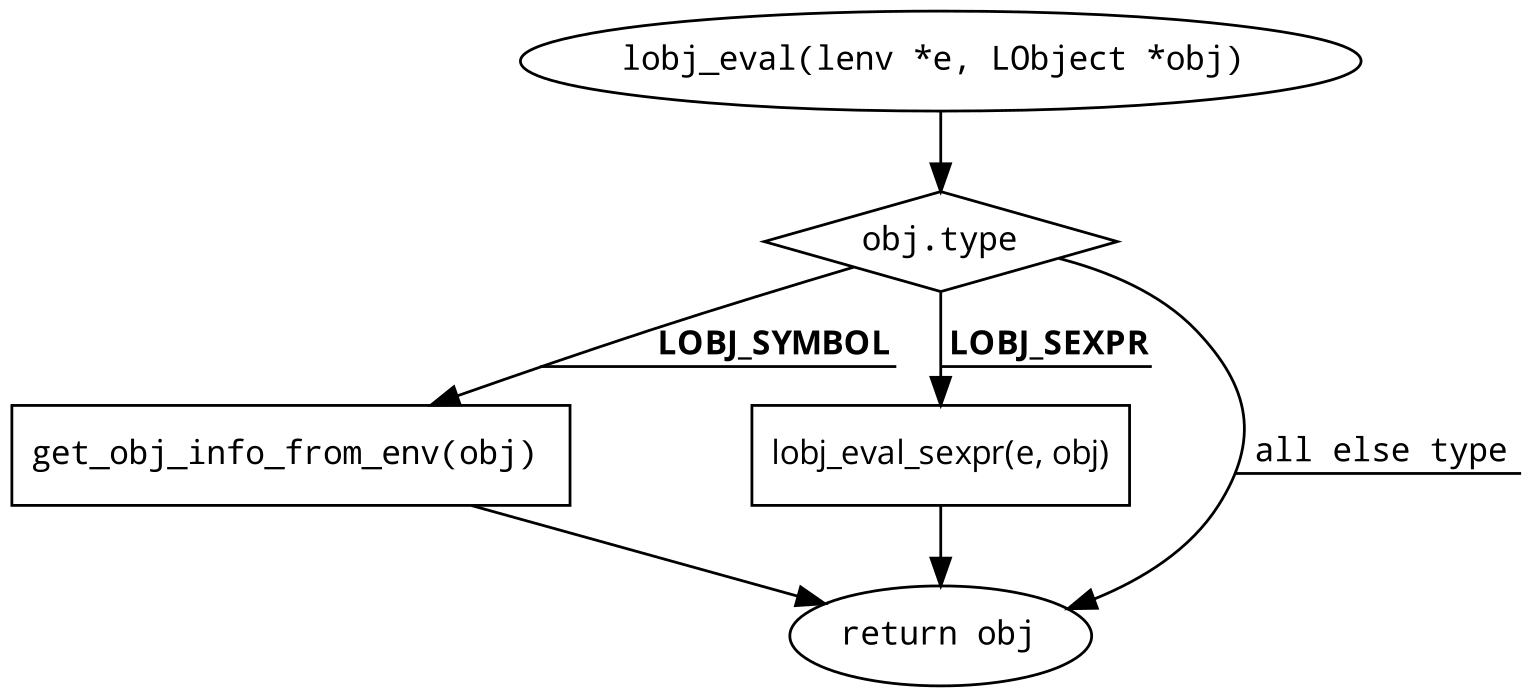
function lobj\_read(tree){  
 switch(tree.tag){  
 case "number","symbol","string":  
 return lobj\_read[number|symbol|string](tree);  
 case ">","sexpr":  
 new result = lobj\_sexpr();  
 break;  
 case "qexpr":  
 new result = lobj\_qexpr();  
 break;  
 for( i in tree.children\_num){  
 skip tokens "[],{},commnet";  
 result = lobj\_add(result, lobj\_read(tree.children[i]));  
 }  
 return result;  
 }  
}



code2flow\_4kCaoH

（当抽象语法树的tag 是“number”、“symbol”或“string”等$$ 规范型时，直接返回含有content 的LObject。

当抽象语法树的tag 是“>”、“sexpr”或“qexpr” 时，需要首先新建一个新的LObject 作为子树树根result ，然后递归调用lobj\_read() 遍历抽象语法树的子树，将结果加入到result 中并返回。）



code2flow\_4kCaoH (1)

## 推导抽象语法树推导

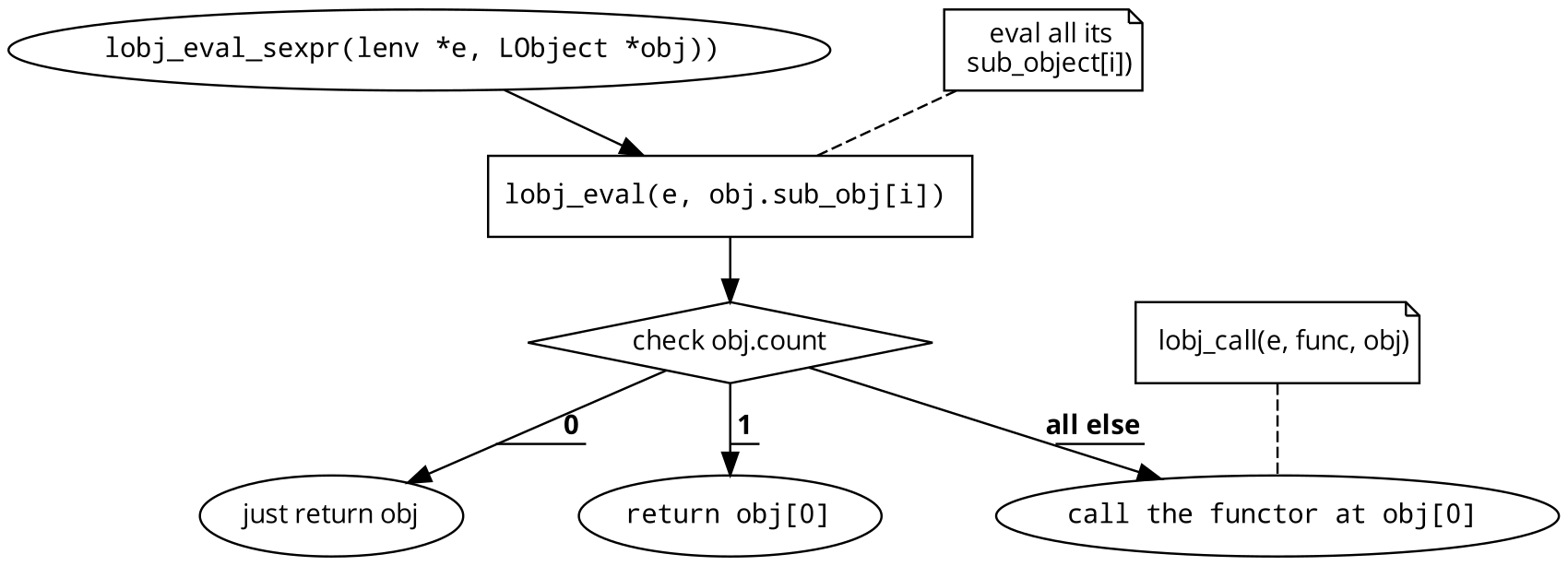
接下来我们要对得到的LObject 树进行推导。只有S-表达式才会被立即推导，S-表达式的符号要从env 环境中获取完整的信息，其余数据类型不做处理。

推导S-表达式的函数是lobj\_eval\_sexpr() ，它会首先递归调用lobj\_eval() 推导其子树，使得树高不断缩减，向上传递结果直到最后只剩一层，最后针对这一层的三种情况依次处理：

| S-表达式的孩子个数 | 类型 | 处理方法 | 举例 |
| --- | --- | --- | --- |
| 0 | 空类型 | 返回自身 | [ ] =>(output): [ ] |
| 1 | 单元素类型 | 返回第一个元素 | [1]=>(output): 1 |
| >=2 | 函数表达式 | 调用相应函数 | [+ 1 1]=> func\_call(e,"+",[+ 1 1])=>(output): 2 |

算法和流程图如图所示：

function lobj\_eval\_sexpr(e, obj){  
 for (i from 0 to obj.count)  
 lobj\_eval(e,obj[i]);  
 switch (obj.count) {  
 case 0: return obj;  
 case 1: return obj[0];  
 }  
 // >=2  
 return lobj\_call(e, func = obj[0], obj);  
}



code2flow\_4kCaoH (2)

## 函数的创建与执行

### 创建预置函数

解释器在开始执行时会初始化一个全局环境变量容器e，然后向其中导入预置函数符号。

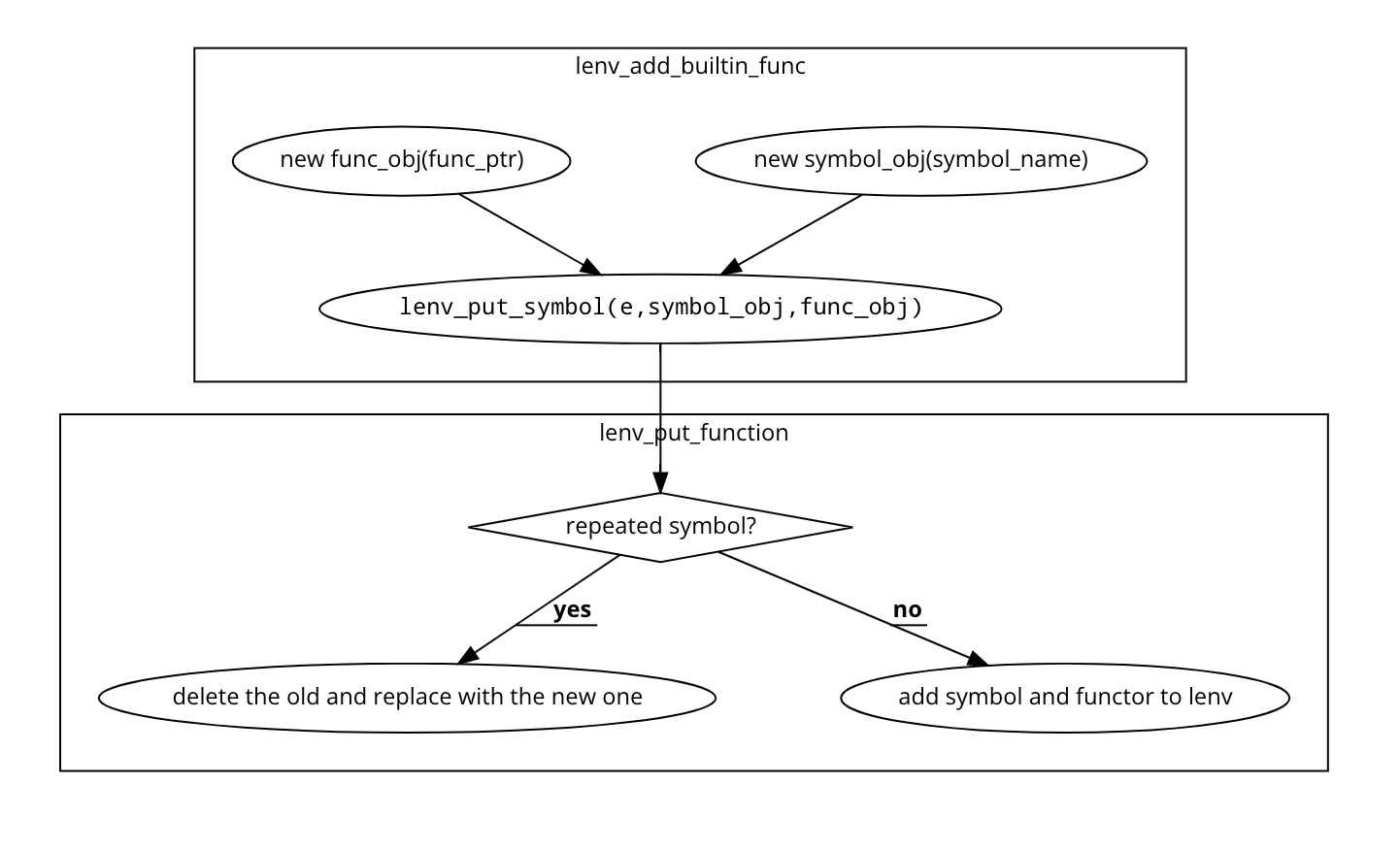
// main.c:   
lenv\* e = lenv\_new();// 初始化环境变量容器  
lenv\_builtin\_init\_list(e);// 初始化符号表

初始化符号表的函数执行一连串的函数添加操作：

// lachesis\_environment.h:  
void lenv\_add\_builtin\_func(lenv\* e, char\* symbol\_name, lbuiltin func);  
// lachesis\_environment.c:  
void lenv\_builtin\_init\_list(lenv \*e)  
{  
 /\*\_\_BASIC\_TYPES\_\_\*/  
 lenv\_add\_builtin\_func(e, "list", built\_in\_list);  
 lenv\_add\_builtin\_func(e, "head", built\_in\_head);  
 lenv\_add\_builtin\_func(e, "tail", built\_in\_tail);  
 lenv\_add\_builtin\_func(e, "eval", built\_in\_eval);  
 ……  
}

添加函数符号的方法的算法和工作流程图如下：

function lenv\_add\_builtin\_func(e, symbol\_name, func) {  
 new symbol\_obj,func\_obj;  
 lenv\_put\_symbol(e,symbol\_obj,func\_obj);  
}  
function lenv\_put\_function(e,symbol\_obj,func\_obj){  
 for (i from 0 to e.count)  
 if (symbol\_repeated)  
 delete the old and replace with the new one;  
 // not repeated  
 add symbol and functor to e;  
}



Screen Shot 2022-04-27 at 11.25.49 AM

首先，字符串 symbol\_name 和函数指针 func 将会被转化成字符对象和函数对象。如果符号已经在环境容器中存在了，那么只要用新的字符对象和函子来替换旧的就可以了。如果是新出现的符号，则需要对环境容器进行扩容然后添加。

### 全局变量和局部变量

两个预置函数被用来实现变量绑定的功能，它们在全局环境变量中的签名如下：

lenv\_add\_builtin\_func(e, "def", built\_in\_define);  
lenv\_add\_builtin\_func(e, "=", built\_in\_put);

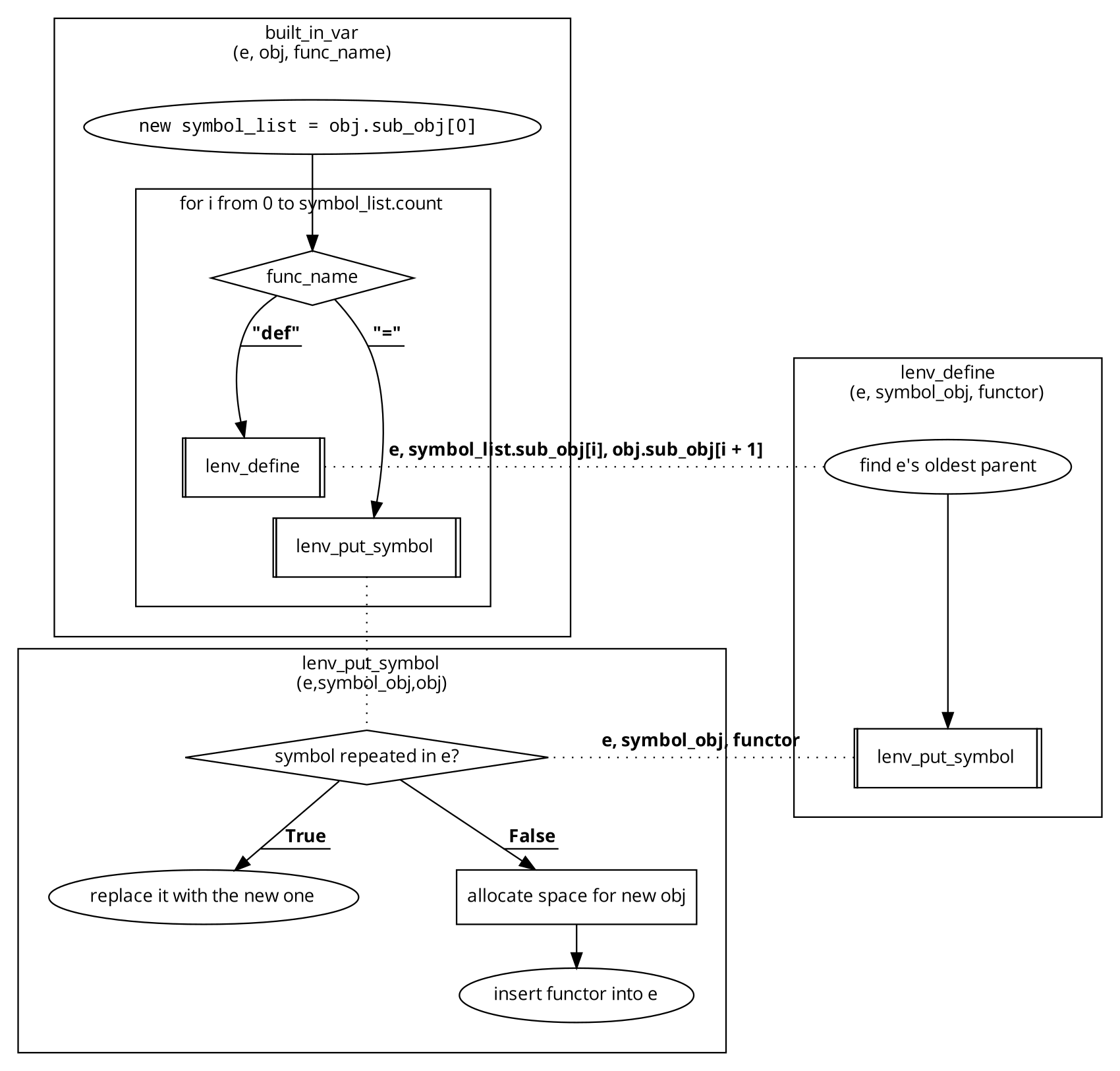
其中，def会把每个符号定义放进全局环境变量中，而=则会把符号定义放在该函数内部的子环境变量中，无法从外部调用。

def 和 = 的实现共享了大部分的代码，在后期重构的过程中，两者被设计成调用一个公共的built\_in\_var函数：

LObject \*built\_in\_put(lenv \*e, LObject \*obj)  
 return built\_in\_var(e, obj, "=");  
  
LObject \*built\_in\_define(lenv \*e, LObject \*obj)  
 return built\_in\_var(e, obj, "def");

built\_in\_var的算法和流程图如下：

function lenv\_put\_symbol(e,symbol\_obj,obj){  
 if(symbol repeated in e){  
 replace it with the new one;  
 }else{  
 allocate space for new obj;  
 insert functor into e;  
 }  
}  
function lenv\_define(e, symbol\_obj, functor) {  
 find e's oldest parent;  
 call lenv\_put\_symbol(e, symbol\_obj, functor);  
}  
function built\_in\_var(e, obj, func\_name){  
 new symbol\_list = obj.sub\_obj[0];  
 for i from 0 to symbol\_list.count{  
 switch (func\_name){  
 case "def":  
 call lenv\_define(e, symbol\_list.sub\_obj[i], obj.sub\_obj[i + 1]);  
 break;  
 case "=" :  
 call lenv\_put\_symbol(e, symbol\_list.sub\_obj[i], obj.sub\_obj[i + 1]);  
 break;  
 }  
 }  
}



code2flow\_7mKd5o

我们以 def {a b c} {1 2 3} 为例解释

### 创建lambda 函数

lambda 是一个特殊的预置函数，它的功能是实现解释器用户自定义的匿名函数。

[[\ {x y} {+ y x}] 1 2 ] ;第一个容器是符号为"\"的函数对象，第二和第三个是参数。

（注意C语言字符串的escape symbol默认是”\“，所以在字符串中表达反斜杠应该写成”\\“）

lenv\_add\_builtin\_func(e, "\\", built\_in\_lambda);

算法如下：

function built\_in\_lambda(e, obj){  
 check obj.sub\_obj[0] is qexpr and only contains symbol;  
 new argument = lobj\_pop(obj, 0);  
 new body = lobj\_pop(obj, 0);  
 return lobj\_lambda(argument, body);  
}  
function lobj\_lambda(argument, body){  
 new func\_obj.type = LOBJ\_FUNC;  
 func\_obj.builtin\_func = NULL; // this means user-define function  
 func\_obj.env = lenv\_new(); // each lambda has its own env  
 func\_obj.argument = arguments;  
 func\_obj.body = body;  
 return func\_obj;  
}

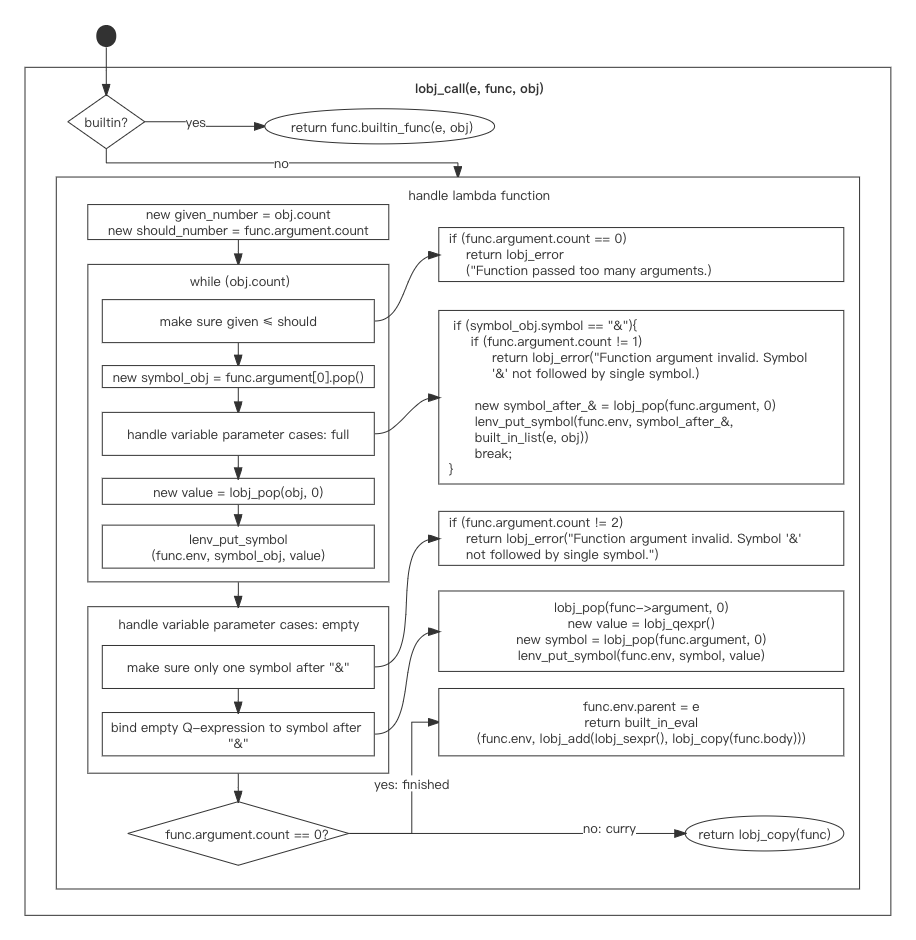
可以看出，built\_in\_lambda和 lobj\_lambda 的功能仅仅是将函数的参数和函数体分别包装进一个函数对象，函数执行的关键功能放在lobj\_call(e, func, obj) 中，由其在运行时捕获符号”\” 后开始工作。

### 执行函数

lobj\_call 的实现相当复杂，难点在于它要完成以下几个功能：

1. 区分预置函数符号（编译时确定）和自建函数符号（运行时确定）的执行。
2. 检查实际给定参数个数和函数接口是否符合。
3. 实现可变参数& 的功能。

其算法和流程图如下：



未命名文件 (3)

function lobj\_call(e, func, obj){  
 //builtin cases:  
 if (func.builtin\_func)  
 return func.builtin\_func(e, obj)  
 //lambda function cases:  
 new given\_number = obj.count   
 new should\_number = func.argument.count  
   
 while (obj.count){  
 if (func.argument.count == 0)  
 return lobj\_error("Function passed too many arguments".)  
 new symbol\_obj = lobj\_pop(func.argument, 0)   
 if (symbol\_obj.symbol == "&"){  
 if (func.argument.count != 1)  
 return lobj\_error("Function argument invalid. Symbol '&' not "  
 "followed by single symbol.")  
 new symbol\_after\_& = lobj\_pop(func.argument, 0)   
 // save the rest obj parameter into a list  
 lenv\_put\_symbol(func.env, symbol\_after\_&, built\_in\_list(e, obj))  
 break  
 }  
 new value = lobj\_pop(obj, 0) // regular cases  
 lenv\_put\_symbol(func.env, symbol\_obj, value) // make it a pair  
 }  
   
 if (func.argument.count > 0 && func.argument.sub\_obj[0].symbol == "&"){  
 if (func.argument.count != 2)  
 return lobj\_error("Function argument invalid. Symbol '&' not "  
 "followed by single symbol.")  
 lobj\_pop(func->argument, 0) // "&"  
 new symbol = lobj\_pop(func.argument, 0)  
 new value = lobj\_qexpr()  
 lenv\_put\_symbol(func.env, symbol, value)  
 }  
   
 if (func.argument.count == 0) { // all finished  
 func.env.parent = e  
 return built\_in\_eval(  
 func.env, lobj\_add(lobj\_sexpr(), lobj\_copy(func.body)))  
 }  
 else // curry cases   
 return lobj\_copy(func)  
}

### 开发新的预置函数

所有的预置函数都遵循函数指针LObject \*(\*lbuiltin)(lenv \*, LObject \*) 的接口，声明在lachesis\_builtin.h中，而具体实现在lachesis\_builtin.c中。

要给lachesis 添加新的预置函数，首先要在lachesis\_environment.c 的符号初始化表函数lenv\_builtin\_init\_list(lenv \*e)中注册新函数的绑定符号和函数指针名，接下来在lachesis\_builtin.h和lachesis\_builtin.c中实现具体的功能。

我们以

## 错误处理

C 语言是一门没有垃圾回收器的静态语言，这意味着在运行时的发生任何内存错误都可能使程序强制退出。因此，一个能在运行时动态检测并处理错误的系统是非常必要的。

在Lachesis 解释器中，我们设计了一套基于宏替换的异常处理系统，用来检查输入参数的语义合法性。

语法分析器保证了输入数据形式上的合法性（例如满足一个S-表达式的形式），但对于我们自定义函数的一些语义上的错误（例如函数接收参数的个数不足）则无法通过语法分析器检出。

为此，我们定义了一套基于宏替换的异常处理系统，在各个功能模块中插入代码块检查运行时的数据：

// base macro  
#define ERROW\_CHECK(args, conditon, fmt, ...) \  
 if (!(conditon)) { \  
 LObject\* err = lobj\_error(fmt, ##\_\_VA\_ARGS\_\_); \  
 lobj\_del(args); \  
 return err; \  
 }  
#endif  
  
// wrapper  
#define ERROW\_CHECK\_TYPE(func, args, index, expect) \  
 ERROW\_CHECK(args, args->sub\_obj[index]->type == expect, \  
 "function '%s' passed incorrect type for argument %i.\nGot %s, " \  
 "Expect %s.", \  
 func, index, lobj\_type\_name(args->sub\_obj[index]->type), \  
 lobj\_type\_name(expect))  
  
#define ERROW\_CHECK\_NUM(func, args, num) \  
 ERROW\_CHECK(args, args->count == num, \  
 "Function '%s' passed incorrect number of argument.\nGot %i, Expect "\  
 "%i", \  
 func, args->count, num)  
  
#define ERROW\_CHECK\_NOT\_EMPTY(func, args, index) \  
 ERROW\_CHECK(args, args->sub\_obj[index]->count != 0, \  
 "Function '%s' passed {} for argument %i.", func, index)

这套系统的核心是基于可变参数的ERROW\_CHECK 函数宏，可变参数 ##\_\_VA\_ARGS\_\_ 依据情况动态地添加可变参数前面的逗号。

其余三条宏都是基于ERROW\_CHECK 的包装器。

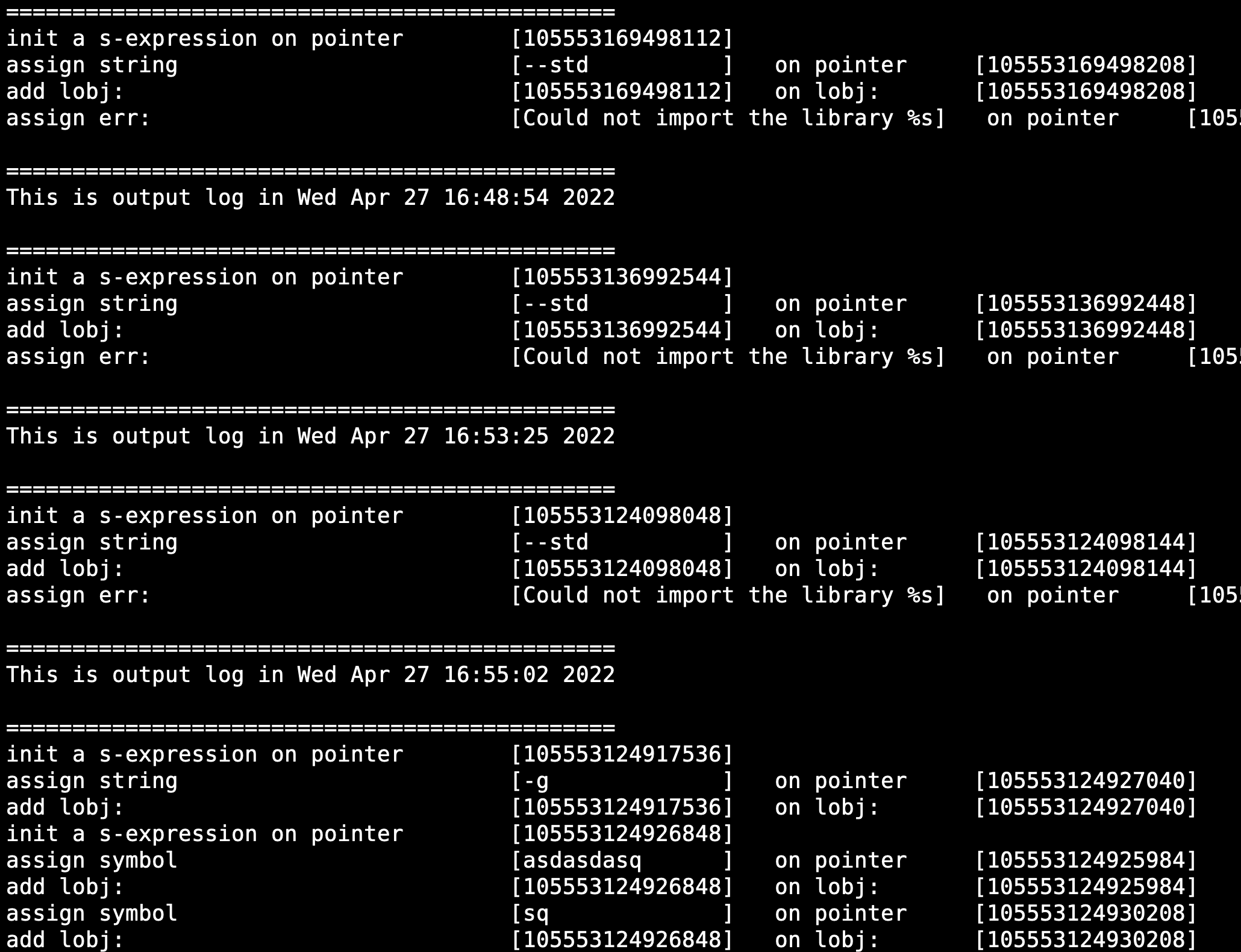
ERROW\_CHECK\_TYPE 限定了 func函数中传入参数 args 的第index 个子对象属性，例如预置的lambda 函数要求第一个和第二个参数都必须是Q-表达式：

LObject\* built\_in\_lambda(lenv\* e, LObject\* obj)  
{  
 ERROW\_CHECK\_TYPE("\\", obj, 0, LOBJ\_QEXPR);  
 ERROW\_CHECK\_TYPE("\\", obj, 1, LOBJ\_QEXPR);

ERROW\_CHECK\_NUM 限定了 func函数中传入参数 args 的第index 个子对象属性，例如预置lambda 函数要求第一个和第二个参数都必须是Q-表达式：

typedef char\* va\_list;  
#define va\_start(ap,v) ( ap = (va\_list)(&v) + sizeof(v) )  
#define va\_arg(ap,t) ( \*(t \*)((ap += sizeof(t)) - sizeof(t)) )  
#define va\_end(ap) ( ap = NULL )

## 动态调试器模式



Screen Shot 2022-04-27 at 4.57.25 PM

## 标准库设计

编程语言的标准库是该语言的每种实现中都按例提供的库。

### 从匿名函数到实名函数

我们内置的lambda 函数抽象可以看做一个匿名函数，

[def {func} [\ {func\_name argument} {  
 def [head f] [\ [tail f] b]   
}]]

[func {not x} {- 1 x}]

## 补遗

### 帮助系统

# 系统结果分析

## 测试用例

本系统分别在两台UNXI 机器上进行测试，

make  
clang  
libreadine-dev



IMG\_2711

（在不同UNIX 机器上的运行）

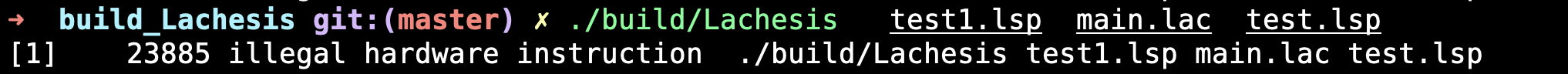
# 结论

## 不足与缺陷

### 神出鬼没内存错误

C/C++ 开发的一大难点就是神出鬼没的内存错误，

### O3优化编译报错



Screen Shot 2022-04-27 at 8.25.04 PM

### 缺少宽字符支持

### import 文件路径逻辑

为了图省事，解释器两种工作模式使用了同一种

### 宏

本系统在开发时使用了一些宏，而在《effective c++》中，避免使用宏的建议被放在第二条。笔者开始不以为意：宏定义是C 语言自由和强大的灵魂，又不是什么洪水猛兽！只要清楚自己在做什么，宏有什么可怕的？

直到在后期不断的系统调试、重构中，我才对宏的的弊端有所了解：

# 致谢

# 参考文献

[1]https://www.phy.cam.ac.uk/blog/how-make-apple-pie-scratch

[mpc]https://github.com/orangeduck/mpc

【functor】https://www.jianshu.com/p/740a3f5c514e

[函子漫画]https://adit.io/posts/2013-04-17-functors,\_applicatives,\_and\_monads\_in\_pictures.html

【函数式编程基础[一]–范畴论和Functor】https://zmxiaodao.com/post/fp/functor/

# 附录