目录

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[1] 指导老师：章林忠

（安徽农业大学 信息与计算科学专业 合肥 230036）

摘要

近些年来，随着 C++、Java、Python 等各大语言纷纷添加对函数式编程的支持，这项历史悠久的技术又焕发生机。函数式编程以函数为程序基本执行单位，打破了传统过程式编程上理念的局限性，使得日趋复杂的软件系统更加易于编写和调试。

本文基于 lambda 演算理论设计了一个简单的函数式语言 Lachesis 以及与之相配套的 lac 解释器系统，并在符合 UNIX 规范的系统上用 C 语言进行了初步的实现与测试。本文实现的解释器分为八个模块，分别是环境管理、内置函数、容器操作、语法解析/词法分析、调试、命令行交互、标准库和日志模块，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。

本文首先介绍了函数式语言相关的理论背景和函数式语言的一般特点，并基于前述需求设计了 Lachesis 语言的核心语法以及实现 lac 解释器系统的总体框架。然后详细说明了 lac 解释器开发的具体技术细节。继而，在随后的章节阐述了 lac 系统的运行环境和对各功能的测试结果。最后，对本文和作者在课题期间的工作成果进行了总结，并提出了进一步的改进方向。

本文通过对 Lachesis 语言和 lac 解释器的设计与实现，介绍了泛函编程的核心想法，展示了在 UNIX 系统中实现一个新的计算机语言的解释器的一般步骤和思路，可作为未来开发新的计算机语言的参考。

关键词：计算理论；计算机语言；解释器；UNIX 编程；

[1]作者简介：高成志，男，（1999.2.13— ），江苏省常州市新北区人，汉族，2018 年 9 月至 2022 年 6 月在安徽农业大学信息与计算科学专业学习。
论文完成时间：2022 年 5 月 20 日

第一章 引言

1.1 选题背景与意义

函数式编程是具有悠久历史的编程范式。函数式与过程式编程之间的角逐在上个世纪末以 C/C++、Java 为代表的过程式语言的大热而告结。在很长一段时间内，函数式编程语言除了在学术界、工业基础设施和人工智能等少数领域外，几乎无人问津。到 21 世纪初，函数式编程的开源地 MIT 都不再用 Scheme[2]作为编程入门课，转而使用 Python。但就在近几年，函数式编程又有起死回生的迹象：C++11、Java8 都对函数式编程提供了专门的 lambda 语法支持，而 Clojure, Scala 等新一代函数式语言也越来越多的用于 Web 开发[1]。

本文设计与实现了一门函数式语言 Lachesis[3]以及配套的 lac 解释器系统。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。同时又有着较强的可拓展性，方便后来的开发者和用户添加新的功能和函数。

通过开发 lac 解释器系统，能够从底层具体实现的角度深入理解泛函编程的核心思想，提升 UNIX 系统编程能力以及对一般动态类型[4]语言的理解。

1.2 国内外研究现状

北京大学马希文教授的《Introduction to Theoretical Computer Science》从数理逻辑上详细阐释了函数式编程背后的计算理论。其另一本著作《LISP 语言》也是入门 LISP 这门经典的函数式语言的经典教材[5,7]。

Harold Abelson、Gerald Jay Sussman 和 Julie Sussman 久负盛名的《Structure and Interpretation of Computer Programs》长年被选为全美计算机类新生入门课程的标准教材。这是一本关于计算机程序设计的总体性观念的基础读物，其用一门函数式语言 Scheme 演示了如何用一些最小概念一步步搭起计算机世界的摩天大厦[8]。

Matthias Felleisen, Robert Bruce Findler 和 Matthew Flatt 的《Semantics Engineering with PLT Redex》是关于计算机语义学的一本实用设计与测试手册，其提供了一个原型工具套件 PLT Redex，用于开发、探索、测试、调试和发布编程语言的语义模型。全书的后半段关于 PLT Redex 的章节为具体实现一个函数式语言解释器提供了蓝本。[10]

---

[2]一门函数式语言。

[3]Lachesis（希腊语：Λάχεσις，意为“命运分配者”）古希腊神话的命运三女神之一。

[4] 动态类型语言是指在运行期间才去做数据类型检查的语言。

Franklin Risby 的《Professor Franklin's Mostly Adequate To Functional Programming》用 JavaScript 介绍了泛函编程的一些常见误区并提出了一些具有针对性的实际编程技巧。[6]

1.3 本文结构

本文共有六章，各章节的主要内容如下：

第 1 章主要介绍了论文的写作背景、国内外的研究现状以及本文的结构。

第 2 章主要介绍了函数式语言相关的理论基础。

第 3 章主要介绍了 Lachesis 语言的特性。

第 4 章主要介绍了 lac 解释器的设计思想和整体框架。

第 5 章具体展示了解释器的实现细节。

第 6 章是本文的总结以及对项目的反思。

第二章 相关理论综述

2.1 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诸多理论模型，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，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。

克服这个限制的一种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。为了阐明这一点，本文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。

2.2 lambda 演算

为了对函数这一抽象向的概念做更精确的形式化表达，数学家阿隆佐·邱奇（Alonzo Church）在 1930 年代设计出了一个能用来表示函数本质的最小逻辑系统，即 Lambda 演算。

最基本的 Lambda 演算系统包括：

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

以下三个规则定义了 lambda 项的基本形式：

表 1 lambda 演算项的定义表

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

或者用更简洁的巴科斯范式[5]表示为：

$$\begin{aligned} M,N,L &= X \\ &\mid (\lambda X.M) \\ &\mid (MN) \\ X &= \text{a variable: } x,y,\ldots \end{aligned} \tag{1}$$

重复上述三个规则的应用就可以递归地得到所有的λ项。

Lambda 的规约操作有两种：

表 2 lambda 演算规约操作的定义表

操作	名称	描述
(λ x.M[x]) -> (λ y.M[y])	α转换	重命名表达式中的绑定变量，用于防止命名冲突。
((λ x.M) E) -> (M[x:=E])	β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（例如本文在解释器的实现中巧妙地做到的那样），那么α转换就不是必须的。而不断运用β规约所得到的最终结果将是一个β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。以下使用布尔代数的实例具体阐述 lambda 演算是如何编码运算的：

布尔代数是一组元素（通常为两个，“TRUE”与“FALSE”）和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[3]，具体来说就是：

$$TRUE \quad := \quad \lambda x.\lambda y.x \tag{2}$$

$$FALSE \quad := \quad \lambda x.\lambda y.y \tag{3}$$

[5]巴科斯范式（英语：Backus Normal Form，缩写为 BNF），是一种用于表示上下文无关文法的语言。它是由约翰·巴科斯（John Backus）和彼得·诺尔（Peter Naur）首先引入的用来描述计算机语言语法的符号集。

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显得抽象。事实上，这个定义是为了满足了布尔型的最小条件，即集合论断言子集A和它的补集A^C的交集为空集：

$$A \cap (A^C) = \emptyset \tag{4}$$

从这个角度看，C 语言通过宏来定义布尔型的方式则是不严谨的，因为{1}的补集不是{0}，而是除了 1 之外的所有整数的集合。[13]

接下来可以编码布尔代数上的运算，例如非（NOT）定义为：

$$NOT \quad := \quad \lambda a.a \quad FALSE \quad TRUE \tag{5}$$

如果使用非进行运算，接收 TRUE 作为参数，计算过程如下：

$$\begin{aligned} \overbrace{\overbrace{NOT}^{\text{函数}}\overbrace{TRUE}^{\text{参数}}}^{} &= \overbrace{\overbrace{(\lambda b.bFALSETRUE)}^{\text{函数}}\overbrace{TRUE}^{\text{参数}}}^{} \\ &= (TRUE)\ \text{FALSE TRUE} \\ &= (\lambda x.\lambda y.x)\ \text{FALSE TRUE} \\ &= FASLE \end{aligned}$$

其结果是一个β规范型，且符合通常对布尔非运算的一般认识。

总而言之，lambda 演算理论提供了一种机制和信心，可以只使用函数计算来解决其他指令编程语言能解决的问题。

2.3 函数式编程

相对于人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。其特点有：

1. 函数是”一等公民”：与其他数据类型处于平等地位，无需特殊操作即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。

2. 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。
3. 函数没有“副作用”：不改变系统内部的状态而仅仅返回一个新的值。

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”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。在计算的最终阶段（例如 2 * 3）依然使用 lambda 演算系统是十分不明智的。

Lambda 演算模型表示的乘法计算过程：

$$
\begin{aligned}
2*3 = MULTIPLY\ 2\ 3 &\Rightarrow \overbrace{(\lambda abc\cdot a(bc))(\lambda sz.s(s(z)))(\lambda xy\cdot x(x(x(y))))}^{\text{MULTIPLY}} \\
&= \lambda c.(\lambda sz.s(s(z)))((\lambda xy\cdot x(x(x(y))))c) \\
&= \lambda cz\cdot((\lambda xy\cdot x(x(x(y))))c)(((xy\cdot x(x(x(y))))c)(z)) \\
&= \lambda cz\cdot(\lambda y\cdot c(c(c(y))))(c(c(c(z)))) \\
&= \lambda cz\cdot c(c(c(c(c(c(z)))))) \\
&= 6
\end{aligned}
$$

图灵机模型表示的乘法计算过程：

$$
\begin{aligned}
2*3 &= MUL\ 00010\ 00011 \\
&= \cdots \text{经过硬件加速的原码一位乘法} \\
&= 6
\end{aligned}
$$

因此，尽管泛函编程最重要的基础是 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，本文的函数式语言解释器是在用指令式的工具表达函数式的思想。

第三章 Lachesis 语言设计

3.1 总论

定义一种语言的一种方法是编写标准规范文档，例如 ECMAScrip[6]，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，否则它就不可能被建造出来。

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。

在 Lachesis 中，这个元语言就是 C，它是大多数 UNIX 系统开发的“母语”。本文将用 C 语言开发一个解释器来解释 Lachesis 语言设计的具体细节，但在这之前，简要列举 Lachesis 语言的核心语法是有必要的。

3.2 S-表达式

表达式求值是计算机程序语言要解决的基本问题之一，例如常见的加法运算：

```
(_input) >> 9 + 1
(output) >> 10
```

事实上，鉴于绝大多数函数抽象语法树采用的是波兰表达式的记号方法。为了方便实现表达式的推导，Lachesis 中的表达式也被设计成以波兰表示法的形式，也就是操作符置于操作数的前面的表示方法。

```
(_input) >> + 1 6
(output) >> 7
```

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：

[6]ECMAScrip 是 JavaScript 的国际标准规范。

```
(_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（英文：SymbolicExpression，S-表达式）。

```
(_input) >> [* + [1 1 3] 2]
(output) >> 10
```

S-表达式不是 Lachesis 独有的，其作为一种以人类可读的文本形式表达半结构化数据的约定在约翰·麦卡锡（John McCarthy）于 1960 年发表的现代函数式语言开山之作《递回函数的符号表达式以及由机器运算的方式，第一部》中得到推广，并以其在 Lisp 家族的编程语言中的使用而为人所知。

S-表达式是 Lachesis 可执行语句的基本单位。它可以只含有一个值，仅仅返回自身。当 S-表达式含有多个值时则对应了 lambda 项中函数应用的概念，此时调用以第一个符号为名字的函数，把其余值作为参数传进去并返回结果。这个过程是递归执行的，这意味着 S-表达式中的每个元素都可以是另一个 S-表达式，一层层嵌套下去。

3.3 Q-表达式

S-表达式一旦被读入就会开始求值，这导致了很多局限。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，在这种情况下希望延迟表达式的推导而仅仅保留其形式（惰性求值）的需求是很自然的，这时就需要引入引用表达式（英文：Quoted-Expression，也译为 Q-表达式）的概念。

Q-表达式和 S-表达式在形式上唯一的不同是其用一对花括号作界定符：

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

Q-表达式惰性求值的特性广泛被运用在程序间传递信息，因为它不像 S-表达式那样会立刻推导，导致表达式内信息的丢失。

3.4 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。再次重复定义符号会重新分配内存。

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

```
(_input) >> def {_var_name} 1
(_input) >> _var_name
(output) >> 1
```

Lachesis 的 def 函数可以像 golang 或 Python 那样一次性给多个变量初始化：

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

在函数式语言中，S-表达式也可以是变量，但是 S-表达式要用用 Q-表达式的形式传递，需要推导时再用 eval 函数将 Q-表达式转换成 S-表达式。

```
(_input) >> def {_add_one} {+ x 1}
(_input) >> def {x} 1
(_input) >> eval  _add_one
(output) >> 5
```

有了 lambda 函数和环境的概念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，例如：

```
namespace{
      do [= _first 1] [= _second 2] [print _first _second]
}
```

其中，_first，_second 都是在函数内部生成的变量名，不会暴露给全局命名空间。

3.5 lambda 函数

lambda 函数是函数式语言通用的核心概念，它是指一类无需定义标识符（函数名）的函数，在 Lachesis 中它是用来定义函数的函数。

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 函数在形式上是一个以“\”为函数名的 S-表达式，其一个 Q-表达式作为参数列表，另一个 Q -表达式作为函数体。每个 lambda 函数都自成一个域，其内部变量对外部不可见。

Lambda 函数的参数个数是可变的，以&为标识符，其后跟一个符号，任意个实际参数都会以 Q-表达式的形式附在这个符号上。

例如：

```
(_input) >> def {&exp}[\ {_operator & _list} {eval [join {_operator } _list]}]
(_input) >> &exp + 1 2 3
(output) >> 6
```

Lambda 函数的参数个数也可以少于指定个数，即可以将原函数转换成一个偏函数[7]。

例如：

```
(_input) >> def  {partial_func} [\ {x y z} {+ x y z}]
(_input) >> def {exp} [partial_func 1 2]

(_input)>> exp
(output) >> [\ {z} {+ x y z}]

(_input)>> exp 2
(output) >> 5
```

[7] 偏函数是指固定一个函数的一些参数，然后产生另一个更小元的函数

3.6 实用方法

以上就是 Lachesis 的核心语法概念，Lachesis 还定义了一些预置的实用函数，列举如下：

表 3 Lachesis 预置函数的定义表

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>	合并多个 Q-表达式	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"
loop <num><qexpr>	循环执行 num 次命令	Loop 2 {print 1} => 1 1

第四章 解释器设计

4.1 工作模式

Lac 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（英文：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。

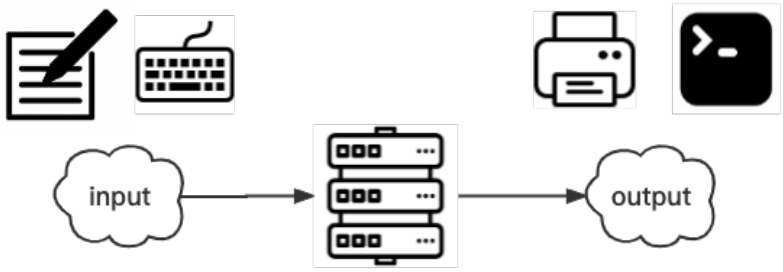

图 1 Lac 解释器的工作模式（1）

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。

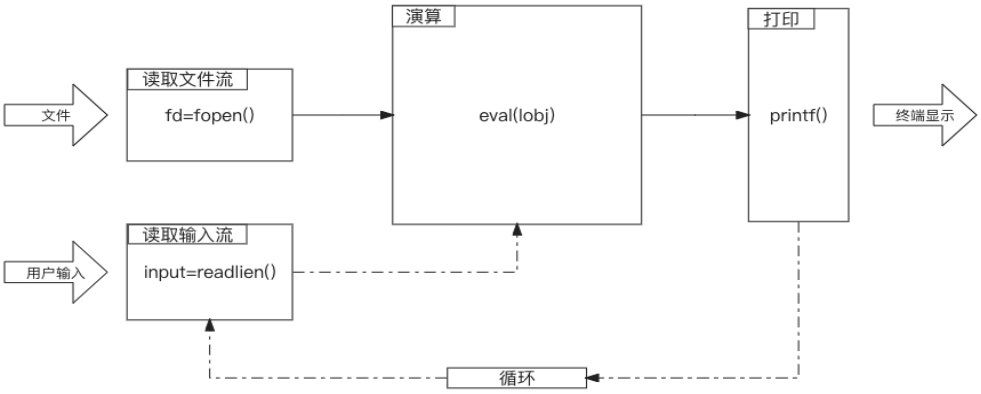

图 2 Lac 解释器的工作模式（2）

4.2 系统框架

lac 解释器的总体结构如图 3 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。

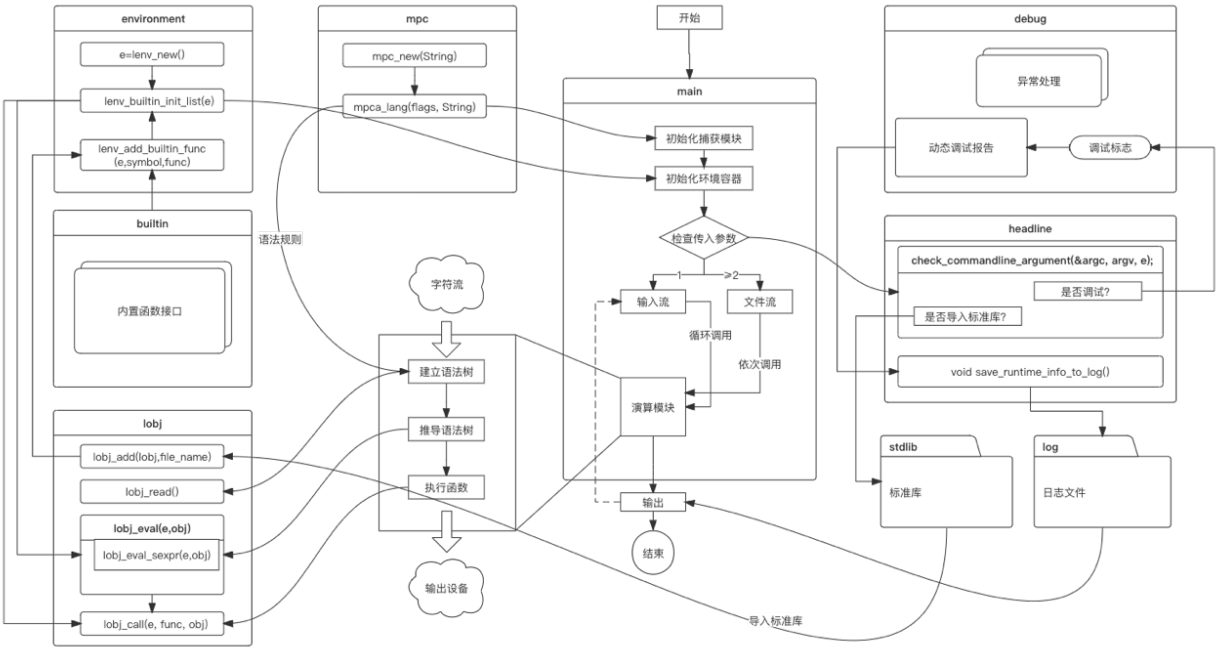

图 3 lac 解释器的总体结构

整个系统的运行流程可以抽象为不断从环境中提取符号定义交由演算器执行函数的过程。图中各模块功能简要介绍如下：

1. 环境管理模块（environment）：创建和初始化环境容器，添加符号和函数。
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）：捕获输入字符流和建立抽象语法树。
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

第五章 解释器实现细节

5.1 开发环境

本系统在一台 Darwin 内核[8]的 MacBook Pro 上使用 C 语言开发，具体技术和配置介绍如下：

5.1.1 命令行软件

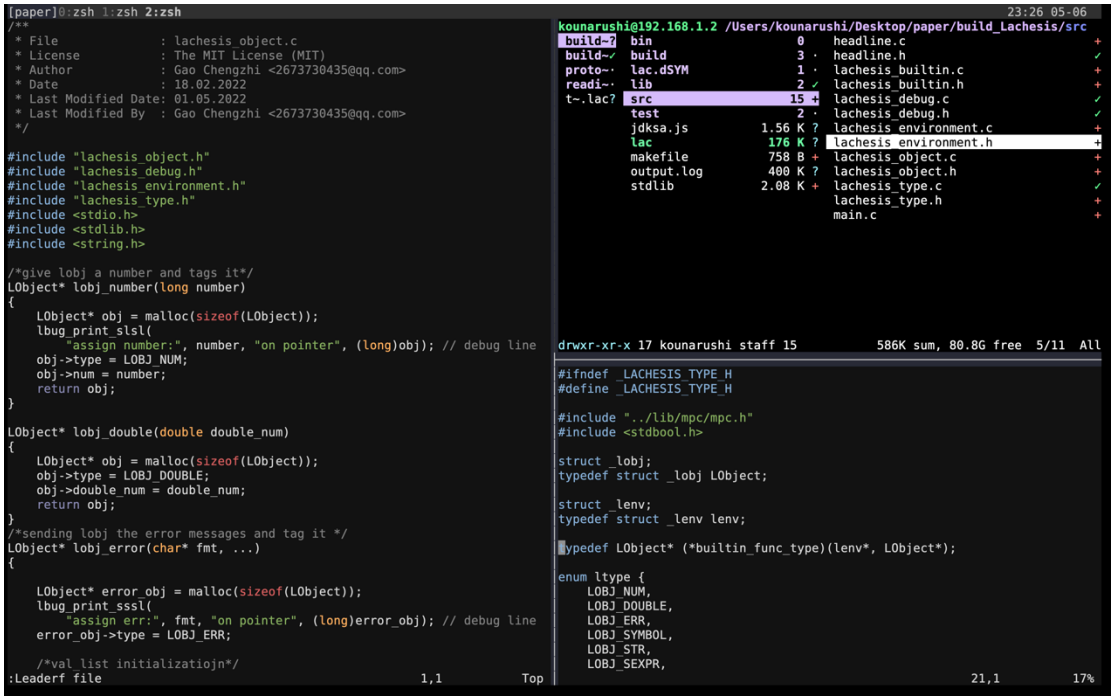

图 4 lac 解释器系统开发的命令行界面

1. vim：终端文本编辑器，本系统主要使用 vim 开发完成，coc.neovim 作为 LSP[9]，LeaderF 作为模糊查找器.
2. 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。
3. 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。

[8]Darwin 是一种类 Unix 操作系统，由苹果公司于 2000 年所发布。它包含开放源代码的 XNU 内核，其以微核心为基础的核心架构来实现 Mach，而操作系统的服务和用户空间工具则以 BSD 为基础。

[9] LSP，即语言服务器协议（Language Server Protocol），是一个开放的、基于 JSON-RPC 的网络传输协议，源代码编辑器或集成开发环境（IDE）与提供特定编程语言特性的服务器之间交互时会用到这个协议。

4. 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。
5. make：编译工具，用于控制项目的构建过程。
6. 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快速打开文件。
7. LLDB：现代 UNIX 调试器，用于单步调试和检查内存错误。

5.1.2 文件路径

本项目的文件结构树安排及其内容如下：

```
build_Lachesis git:(master)
├── bin          // 二进制库依赖
├── build        // 最终产出的可执行二进制目标文件
├── lac          // 临时产出的可执行二进制目标文件
├── lib          // 联合编译的C依赖库
├── makefile     // 编译过程控制脚本
├── output.log   // 调试信息输出文件
├── src          // 开发源文件
├── stdlib       // Lachesis标准库文件
├── test         // 测试样例文件夹
└── README.md    // 项目说明文件
```

图 5 lac 解释器系统开发的文件结构

5.1.3 自动编译脚本

综合考虑 lac 解释器项目的实际需求和规模，本文使用了一个微型的 Makefile 来控制编译过程，一些关键代码解释如下。

首先是定义了一些通过终端工具自动推导的别名：

```
cc = clang
target =i lac
SRCDIR := src
TESTDIR := test
deps = $(shell find  ./src -name "*.h")
src = $(shell find  ./lib -name "*.c"&& find ./src -name "*.c")
testfile = $(shell find  ./test -name "*.lac")
obj = $(src:%.c=%.o)
```

然后用这些别名编写可拓展的编译指令：

```
debug:
   #-g means gdb
```

```
    $(cc) -Wall -g  -lreadline  $(src) -o lac
install:
    $(cc) -Wall -O3 -lreadline  $(src) -o ./build/lac
test:
    ./lac $(testfile)
clean:
    -@rm -f lac
    -@rm -f build/lac
```

使用时以 make [command] 为格式自动化编译指令，例如：

```
➜  build_Lachesis git:(master) ✗ make install debug
clang -Wall -O3 -
lreadline  ./lib/mpc/mpc.c ./src/lachesis_type.c ./src/headline.c ./src/lachesis_buil
tin.c ./src/lachesis_debug.c ./src/main.c ./src/lachesis_environment.c ./src/lachesi
s_object.c -o ./build/lac

#-g means gdb
clang -Wall -g  -
lreadline  ./lib/mpc/mpc.c ./src/lachesis_type.c ./src/headline.c ./src/lachesis_buil
tin.c ./src/lachesis_debug.c ./src/main.c ./src/lachesis_environment.c ./src/lachesi
s_object.c -o lac
```

5.2 容器设计

lac 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。

如图 4 所示，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. 指向环境变量容器的指针和指向存储函数对象信息的指针。

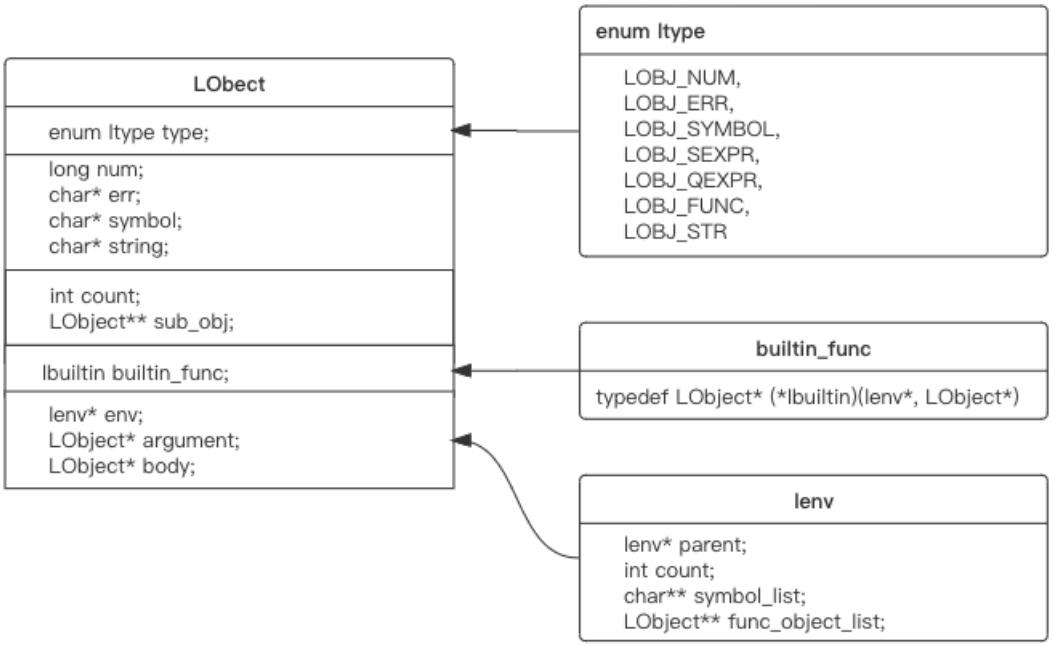

图 6 LObject 容器数据结构示意图

5.3 语法分析器

语法分析器的功能是将输入字符流通过指定的捕获规则过滤，再创建函数抽象语法树。

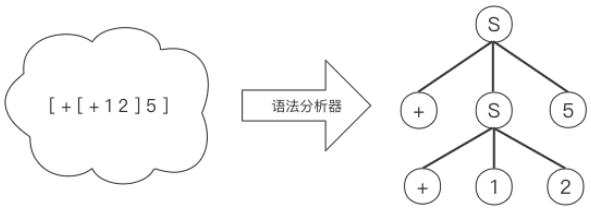

图 7 语法分析器的功能

lac 没有自己开发专用的语法分析析器，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（英文：Micro Parser Combinators，微型解析器组合子），被用来执行 Lachesis 的语法解析工作。

如下表所示，通过调用 mpca_lang()函数，lac 解释器使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。

表 4 lac 解释器的捕获规则表

规则	定义	解释
number	/-?[0-9]+/	可选“-”开头，后跟任意个 0-9 之间的数字
double	/`-?[0-9]*\\.?[0-9]+`/	以`包裹的<number>，中间可选有.的组合
symbol	/[a-zA-Z0-9_+\\-*\\/\\\\=<>!&]+/	任意个字母、数字、_+-* /=<>!&的组合
string	/\"(\\\\.|[^\"])*\"/	以\开头的单字符或除了”之外的任意字符
comment	/;[^\\r\\n]*/	以;开头的除了\r\n 之外的任意多个字符
sexpr	'[' <expr>* ']'	被”[ ]“包裹起来的任意条<expr>
qexpr	'{' <expr>* '}'	被”{ }“包裹起来的任意条<expr>
expr	<number>|<doubel>| <symbol>|<string>| <comment>|<sexpr>|<qexpr>	任意一条以上七种种类型
lexpression	/^/ <expr>* /$/	任意多条<expr>

由上表可知，expr[ession] 只是前七种类型的囊括，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)){……}
```

5.4 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：

```
typedef struct mpc_ast_t {
  char* tag;
  char* contents;
  mpc_state_t state;
  int children_num;
  struct mpc_ast_t** children;
} mpc_ast_t;
```

其中，tag 指示了捕获词符的类型（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。

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。当抽象语法树的 tag 是“number”、“symbol”或“string”等β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。

lobj_read() 的具体算法和流程图如下：

```
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;
}
```

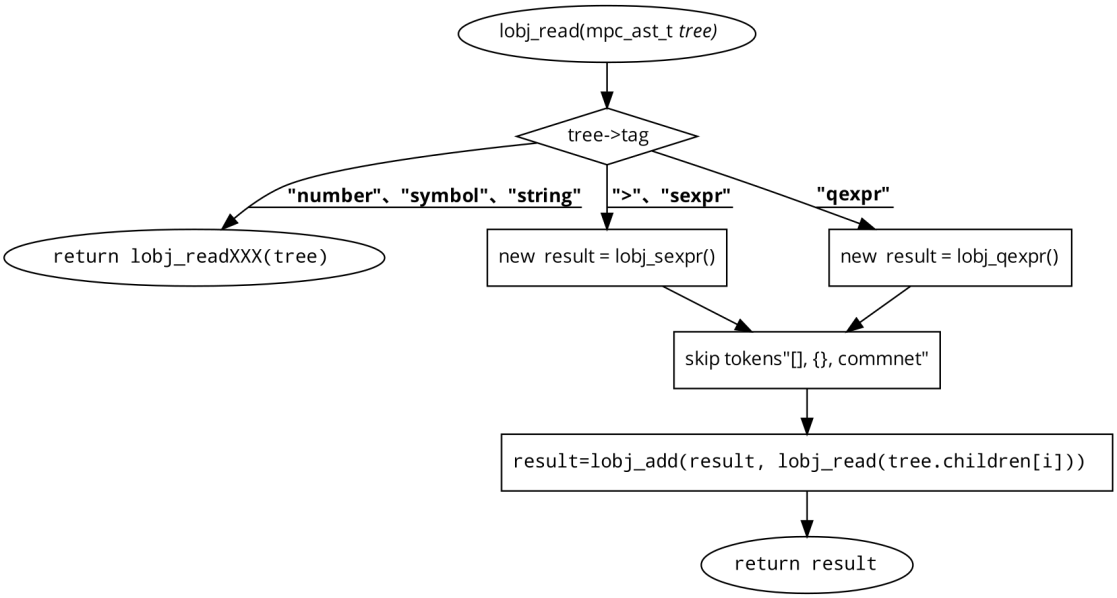

图 8 lobj_read()函数算法流程图

5.5 推导抽象语法树推导

读取并建立抽象语法树之后要对其进行推导。负责推导的是 lobj_eval()函数，其首先检查传入容器的类型，只有 S-表达式才会被立即推导，S-表达式的符号要从 env 环境中提取完整的信息，其余数据类型不做处理。

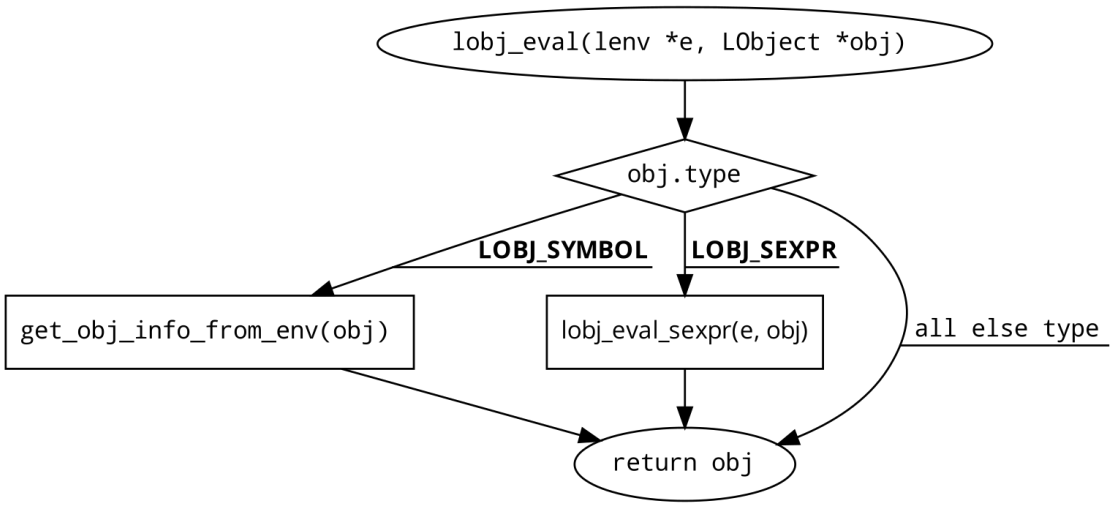

图 9 lobj_eval()函数算法流程图

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：

表 5 lac 解释器的捕获规则表

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])

lobj_eval_sexpr()函数的算法和流程图如下：

```
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);
}
```

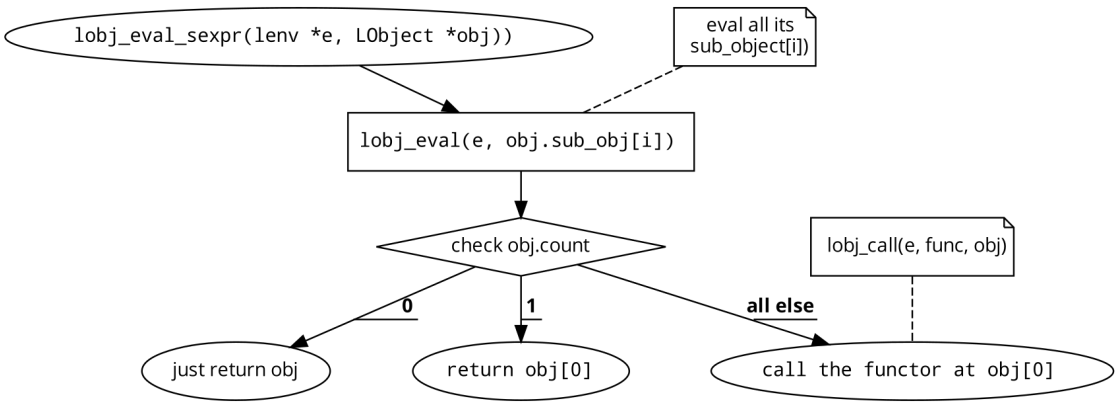

图 10 lobj_eval_sexpr()函数算法流程图

5.6 函数的创建与执行

5.6.1 载入预置函数

lac 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 lenv_add_builtin_func()执行一系列的函数添加操作：

```
// 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);
    ……
}
```

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，则只要用新的字符对象和函数对象来替换旧的。如果是不在环境容器中的符号，则需要对环境容器进行扩容然后添加该符号及其函数对象。

lenv_add_builtin_func()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);
   }
   functionlenv_put_function(e,symbol_obj,func_obj){
   for (i from0 to e.count)
      if (symbol_repeated)
         delete the old and replace with the new one;
      else
         add symbol and functor to e;
   }
}
```

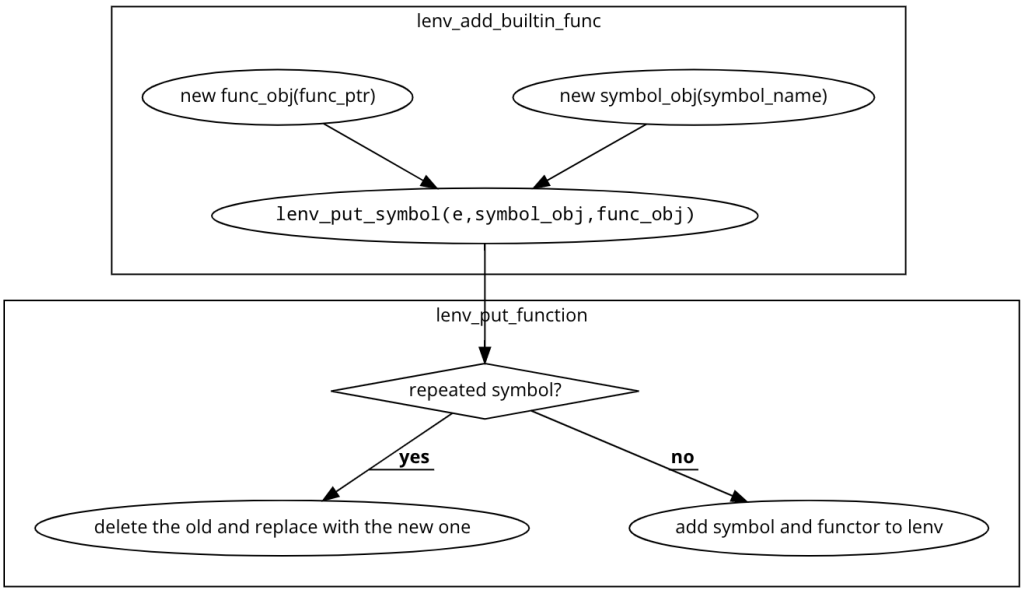

图 11 lenv_add_builtin_func()函数算法流程图

5.6.2 全局变量和局部变量

两个预置函数"def"和"="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 会把每个符号定义添加进全局环境变量中，而=则会把符号定义添加在该函数内部的局部环境变量中。

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 函数来减少冗余：

```
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 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;
    }
  }
}
```

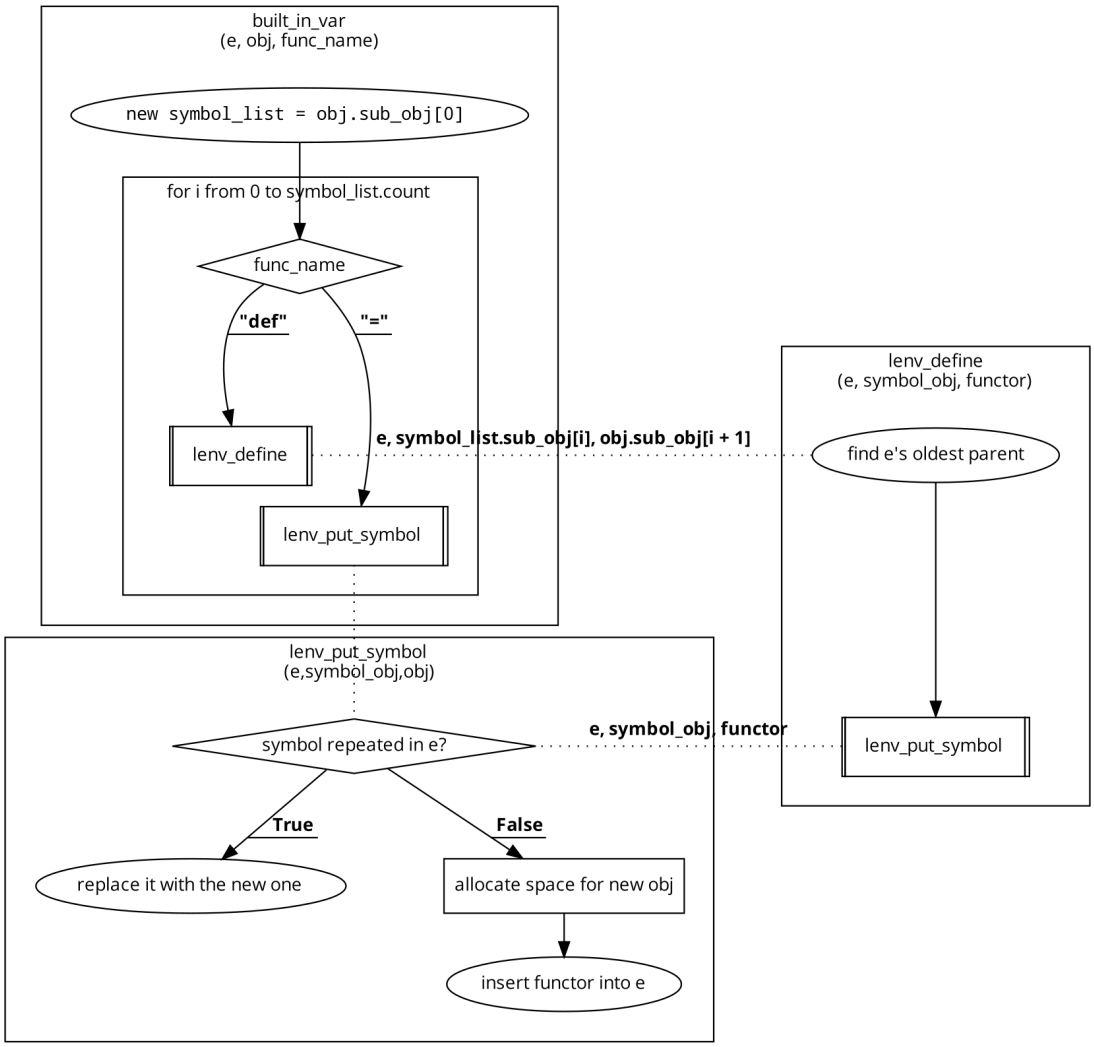

图 13 built_in_var()函数算法流程图

以 def {a b c} 1 2 3 为例解释以上工作流程：首先 obj 的第一个孩子{a b c}被提取出来成为一个 LOBJ_QEXPR 类型的容器 symbol_list。对于 symbol_list 中的每一个符号，都执行了一次 lenv_define(e,symbol_list->sub_obj[i], obj->sub_obj[i + 1]);操作，将对应的参数添加到对应的符号中。

lenv_define()执行的是“def”，即定义一个全局变量的工作。其具体实现如以下代码所示：它要先不断向上回溯，找到全局环境变量所对应的容器，然后调用 lenv_put_symbol()将符号以及对应的内容添加进去。

```
void lenv_define(lenv *e, LObject *symbol_obj, LObject *functor){
/*find e->parent's parent... */
while(e->parent)
        e = e->parent;
```

```
    lenv_put_symbol(e, symbol_obj, functor);
}
```

5.6.3 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。

```
(_input)>>[[\ {x y} {+ y x}] 1 2 ];
(output)>> 3
```

第一个容器是符号为"\"的函数对象，第二和第三个是参数。

（注：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);
    returnlobj_lambda(argument, body);
}

function lobj_lambda(argument, body){
    new func_obj.type= LOBJ_FUNC;
    func_obj.builtin_func= NULL;// this means user-definefunction
    func_obj.env=lenv_new();// each lambda has its own envir
    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。

5.6.4 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）。
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。

第一个问题使用函数对象内部的函数指针的值来判定，如果是自建函数符号，在初始化对象时会给 builtin_func 赋 NULL 值，反之则会指向要调用的预置函数的地址。

第二个问题要仔细对比各种可能出现情况，将实际给定参数一个个弹出绑定到函数接口所给定的变量上。

第三个问题要在&之前的符号都绑定完成之后，将可变参数部分全部连接成一个 Q-表达式赋在&之后的那个符号上。

具体算法和流程图如下：

```
function lobj_call(e, func, obj) {
  //builtin cases:
  if (func.builtin_func)
    return func.builtin_func(e, obj)
  //lambda function cases:
  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)
}
```

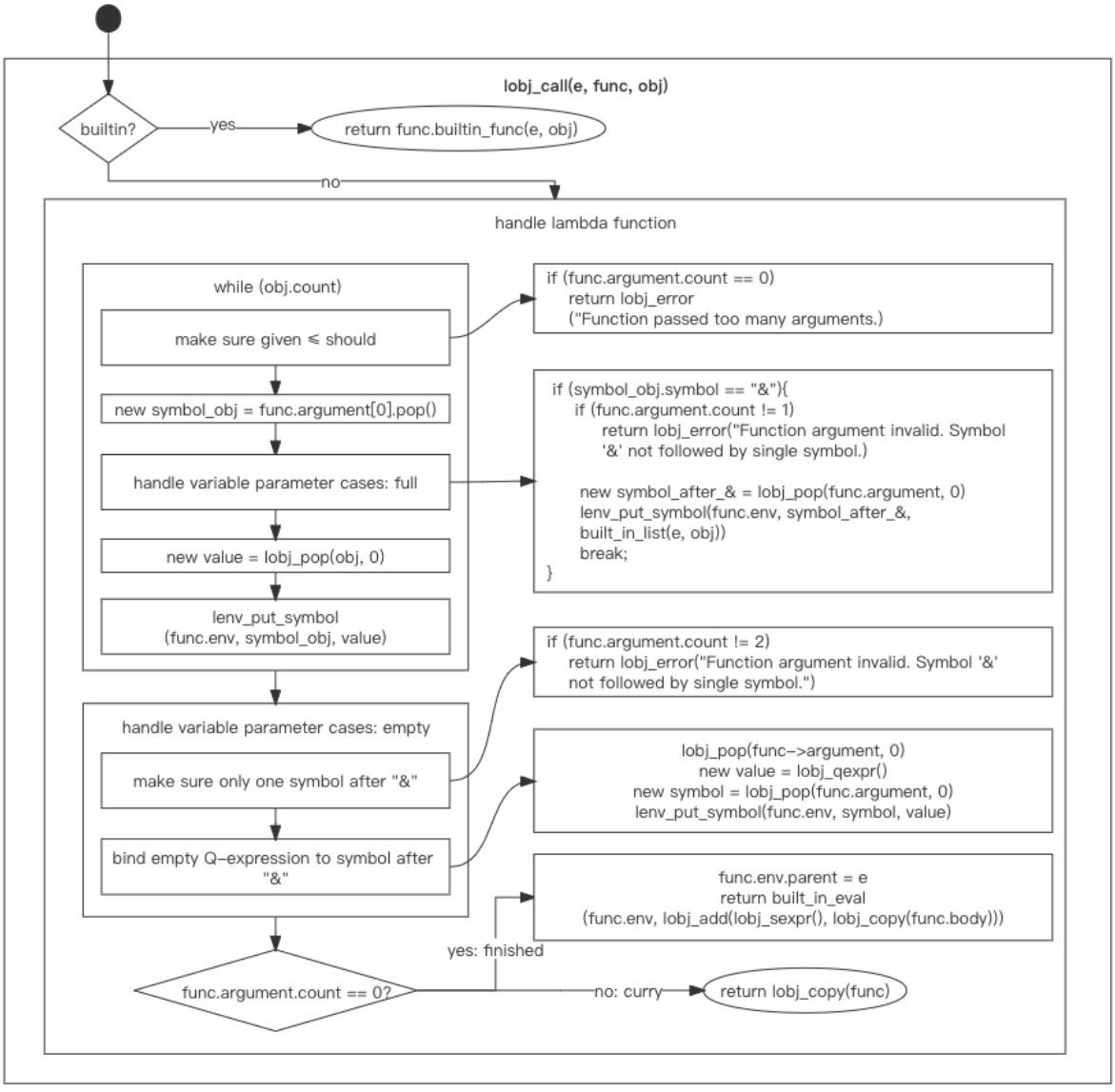

图 14 lobj_call()函数算法流程图

5.7 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 forargument %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 ofargument.\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 的包装宏。

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 函数要求仅允许传入两个参数：

```
LObject* built_in_lambda(lenv* e, LObject* obj){
ERROW_CHECK_NUM("\\", obj, 2);
……
```

ERROW_CHECK_NOT_EMPTY 限定了 func 函数中传入参数 args 个数不能为空：

```
LObject* built_in_head(lenv* e, LObject* obj){
ERROW_CHECK_NOT_EMPTY("head", obj, 0);
……
```

这套系统能有效地在运行期处理语义异常，防止因为语义错误导致的程序崩溃。

```
Lachesis >head {1 2 3} {2 3 4}
Error: Function 'head' passed incorrect number of argument.
Got 2, Expect 1

Lachesis > head [+ 1 2]
Error: function 'head' passed incorrect type for argument 0.
Got Number, Expect Q-expression.

Lachesis > head {}
Error: Function 'head' passed {} for argument 0.
```

图 15 异常处理

5.8 动态调试器模式

为了更好地在开发过程中检查可能出现的各种问题，特别是内存错误，lac 解释器钟设计了一个动态调试器模式。

动态调试器模式通过命令行参数-g 传入，如下：

```
➜build_Lachesisgit:(master) ✗lac -g
```

在 check_commandline_argument()函数中会捕获传入的所有参数并进行处理，当检测到-g 参数时，会首先激活_debug_mode 这个全局变量，然后将一些基本运行时信息写入日志文件。

```
if (argv[i][1] == 'g'){
_debug_mode = true;
    save_runtime_info_to_log();
    return_number = 1;
……
```

当捕获器感知到_debug_mode 开启后会在每次读取抽象语法树时对其进行打印：

```
if (_debug_mode)
    mpc_ast_print(raw.output);
```

打印效果如下：

```
λ -> [== [+ `0.1` `0.2`] `0.3`]
>
  regex
  expr|sexpr|>
    char:1:1 '['
    expr|symbol|regex:1:2 '=='
    expr|sexpr|>
      char:1:5 '['
      expr|symbol|regex:1:6 '+'
      expr|double|regex:1:8 '`0.1`'
      expr|double|regex:1:14 '`0.2`'
      char:1:19 ']'
    expr|double|regex:1:21 '`0.3`'
    char:1:26 ']'
  regex
1
λ ->
```

图 16 抽象语法树打印效果

同时各个初始化函数和析构函数被调用时也会将信息写入日志文件，这有助于我们批量统计哪些容器的内存没有释放。

```
==========================================
This is output log in Tue May  3 10:43:10 2022

==========================================
init a s-expression on pointer        [105553131700224]
add lobj:                             [105553131700320]   on lobj:       [105553131700224]
pop lobj the number:                  [0                ]   on pointer    [105553131700224]
delete lobj:                          [105553131700224]   type           [S-expression   ]
delete lobj:                          [105553131700320]   type           [double         ]

==========================================
This is output log in Tue May  3 20:48:50 2022

==========================================
init a s-expression on pointer        [105553167400960]
assign symbol                         [import         ]   on pointer    [105553167401056]
add lobj:                             [105553167401056]   on lobj:       [105553167400960]
assign string                         [stdlib         ]   on pointer    [105553167401152]
add lobj:                             [105553167401152]   on lobj:       [105553167400960]
delete lobj:                          [105553167401056]   type           [Symbol         ]
pop lobj the number:                  [0              ]   on pointer    [105553167400960]
init a s-expression on pointer        [105553167434016]
init a s-expression on pointer        [105553167433920]
assign symbol                         [def            ]   on pointer    [105553167436416]
add lobj:                             [105553167436416]   on lobj:       [105553167433920]
init a q-expression on pointer        [105553167436512]
assign symbol                         [func           ]   on pointer    [105553167436608]
add lobj:                             [105553167436608]   on lobj:       [105553167436512]
add lobj:                             [105553167436512]   on lobj:       [105553167433920]
init a s-expression on pointer        [105553167436704]
assign symbol                         [\              ]   on pointer    [105553167436800]
add lobj:                             [105553167436800]   on lobj:       [105553167436704]
```

图 17 output.log 文件截图

5.9 标准库设计

5.9.1 总论

编程语言的标准库是使用该语言编写的一系列实用方法，lac 解释器的标准库可以通过-std 命令行参数在任意执行路径导入，也可以在 stdlib 所在路径中用 import “stdlib”导入：

```
➜build_Lachesisgit:(master) ✗lac --std
```

或者

```
[import “stdlib”]
```

函数式语言的标准库的设计是尾递归编程[10]的艺术。lac 解释器的演算模块和环境变量定义方式被设计成支持尾递归和惰性加载[11]的形式，从而能够充分利用递归编程的技术。

5.9.2 命名原则

虽然 lambda 函数内部的变量名不会暴露给外部，但是外部的变量名很容易和内部变量名起冲突。为了解决这个问题，我们遵循最传统的 UNIX C 编程命名原则，将标准库内部的变量用双下划线表示。

5.9.3 从匿名函数到实名函数

Lachesis 内置的 lambda 函数功能实现了一个匿名函数，如果下次使用还需要再绑定到一个变量上。

现在，标准库的 func 函数将这两个步骤打包成一个函数：

```
[def {func} [\ {__name_and_argument __body} {
   def [head __name_and_argument] [\ [tail __name_and_argument] __body]
    }]]
```

func 是一个符号，被绑定在一个匿名函数上。这个匿名函数接收两个参数，第一个是__name_and_argument，顾名思义就是函数名和函数参数，第二个是__body，也就是函数体。首先__name_and_argument 的 head （所以__name_and_argument 必须是一个 Q-表达式）被提取出来作为一个符号被 def 定义为另一个匿名函数，以 tail __name_and_argument 为参数，__body 为函数体（__body 作为函数体也必须是一个 Q-表达式）。

有了新的 func 函数就能用更简洁的方式定义一般函数，例如非函数：

```
[func {not __x} {- 1 __x}]
```

5.9.4 列表操作

对 Q-表达式进行操作的需求是十分旺盛的，尤其是现代 lambda 模块中常常会实现的 map 和 filter 函数：

[10] 尾递归即函数的最后一个操作是返回函数调用的结果。

[11]惰性加载指获得一个属性对象时，这个对应对象实际并没有保存在运行空间中，而是在其读取方法第一次被调用时才从其他数据源中加载到运行空间中。

```
λ -> map [\ {x} {* x x}] {1 2 3 4 5}
{1 4 9 16 25}
λ -> filter [\{x} {> x 3}] {1 2 3 4 5}
{4 5}
```

其具体实现如下：

```
[func {first __lexpr} {eval [head __lexpr]}]
[func {map __operation __list} {
   if[== __list nil]
      {nil}
      {join
         [list [__operation [first __list]]]
         [map __operation [tail __list]]}
   }]
[func {filter __fil __list} {
   if [== __list nil]
      {nil}
      {join
          [if [__fil [first __list]]
            {head __list}
            {nil}]
          [filter __fil [tail __list]]}
   }]
```

它们都接受一个函数作为操作，另一个 Q-表达式作为操作的列表，实现思路也都是 join 处理过的第一个元素以及其余递归后的元素。

本文还定义了一些常见的列表操作函数，举例和具体实现如下：

```
λ -> def {a b c d e f} 1 2 3 4 5 6

λ -> def {_list} {a b c d e f}

λ -> len _list
6
λ -> at 3 _list
4
λ -> last _list
6
λ -> iselem 8 _list
0
```

```
[func {len __list}{
   if [== __list nil]
      {0}
      {+ 1 [len [tail __list]]}
      }]
```

```
[func {last __list} {
   at [- [len __list] 1] __list
   }]
[func {at __n __list} {
   if [== __n 0]
      {first __list}
      {at [- __n 1] [tail __list]}
      }]
[func {iselem __x __list} {
   if [== __list nil]
      {false}
      {if [== _x __list]
         {true}
         {iselem __x [tail __list]}
         }
   }]
[func {join_eval & xs} {eval xs}]
```

5.9.5 数学函数

一些常见的数学函数是十分必要的，但是鉴于时间和笔者捉襟见肘的函数式编程水平，只实现了一些最简单的数学函数。

```
; const define

[def {MATH_E} `2.718`]
[def {MATH_LN2} `0.693`]
[def {MATH_LN10} `2.303`]
[def {MATH_LOG2} `1.443`]
[def {MATH_LOG10} `0.434`]
[def {MATH_PI} `3.141`]
[def {MATH_SQRT2} `1.414`]

; function

[func {abs __num} {
    if [> __num 0]
      {__num}
      {- __num}
    }]

[func {cube __num} {* __num __num __num}]
[func {square __num} {* __num __num}]

[func {range_sum __low __high} {
   if [> __low __high]
      {0}
      {+ __low [range_sum [+ __low 1] __high]}
   }]

[func {exp __base __n} {
```

```
  if [== __n 0]
    {1}
    {* __base [exp __base [- __n 1]]}
  }]

[func {sum_square __base __n} {
  if [== __n 0]
    {1}
    {+ __base [sum_square __base [- __n 1]]}
  }]

[func {sum_cube __base __n} {
  if [== __n 0]
    {1}
    {+ __base [sum_cube __base [- __n 1]]}
  }]

[func {fib __n} {
  if [== __n 0]
    {0}
    {if [== __n 1]
      {1}
      {+
      [fib [- __n 1]]
      [fib [- __n 2]]
      }
    }
  }]
```

第六章 结论

6.1 测试用例

本系统分别在两台 UNXI 机器上进行了测试，系统信息如下图所示：

图 18 在不同 UNIX 机器上的运行

初次运行需要安装以下依赖库：

```
make
clang or gcc
libreadine-dev
```

进入开发文件夹之后用以下命令安装/卸载/调试/测试:

```
>> make install
>> make clean
>> make debug
>> test
```

运行选择可以直接执行二进制文件：

```
(debug_use_) >> ./lac
(regular_use) >> ./build/lac
```

或者将当前路径加入系统执行路径中直接运行：

```
export PATH="当前绝对路径/build_Lachesis/build:$PATH"
(input) >>lac
```

一个用 Lachesis 自身写成的单元测试集合被用来测试程序，结果如下表所示：

表 6 预置函数测试表

测试名称	样例	结果
简单整数加减乘除	+-*/ 1 3	✅结果正确
越界整数加法乘法	+-*/ 1.79E+11 1.79E+13	✅成功捕获异常
结果越界整数加法乘法	+* 21331 1.79E+39	✴️结果溢出，未成功捕获异常，程序正常
简单浮点数加减乘除	+-*/ 1.22 3.23	✅结果正确
输入越界浮点数加法乘法	+-*/ -1.79E+309 +1.79E+309	✴️捕获器不支持科学记数法，手 310 位输后成功捕获异常
结果越界浮点数加法乘法	+* 2133.1 43124324543.2334	✴️结果溢出，未成功捕获异常，程序正常
整数浮点数混合运算	+-*/ 1.23 4	✅结果正确
数字类比较	<>>= <= == != 4 6	✅结果正确
浮点数边界比较	[== [+ 0.1 0.2] 0.3]	✅结果正确
屏幕输出	print “simple sentence!”	✅结果正确
文件导入	import “stdlib”	✅结果正确
屏幕转义字符输出	print “test \0 test test est”	✅结果正确
文件输出	fprint “new.txt” “simple sentence!”	✅结果正确
条件分支	[if [== 1 2] {print “F”} {print “T”}]	✅结果正确
循环	[loop 1000000 {print “long……”}]	✅结果正确
超大循环	[loop 10000000 {print “long……”}]	🆘调用栈溢出，程序崩溃
Y 组合子[12]	[\{x} {x x}][\{x} {x x}]	✅结果正确，递归成功，最后调用栈溢出，程序退出

[12] 在 lambda 演算中最简单的一个不动点组合子，会无限循环且形式不变（不动），是计算其他函数的一个不动点的高阶函数。

表 7 标准库函数测试表

测试名称	样例	结果
实名函数	[func {not __x} {- 1 __x}] [not 1]	✅结果正确
布尔类型	[and/or/ true false]	✅结果正确
列表长度	len {a b v c d}	✅结果正确
列表下标	at 3 { a b c d e}	✅结果正确
列表元素判断	iselem obj {a obj 3 4 “acas”}	✅结果正确
斐波那契数列	fib 12	✅结果正确
绝对值	abs -222	✅结果正确
平方/立方	cube 3/square 3	✅结果正确
求幂	exp 3 6	✅结果正确
平方和/立方和	sum_cube/square 2 5	✅结果正确
等差数列求和	range_sum 1 10	✅结果正确
映射	map [\ {x} {* x x}] {1 2 3 4 5}	✅结果正确
过滤	filter [\{x} {> x 3}] {1 2 3 4 5}	✅结果正确
序列化指令	do [= {x} 1] [print x]	✅结果正确
名字空间	[namespace { do [= {x} 1] [print x] }] x	✅结果正确

6.2 不足与缺陷

6.2.1 缺少对开发者的支持

对开发者来说，Lachesis 是一门非常不友好的语言。错误提示不全面、没有语法着色和自动补全让开发 Lachesis 程序的过程异常艰苦。由于函数式编程的特点，一旦程序出错，唯一的调试方法就是逐行检查代码。

6.2.2 不支持跨平台运行

解释器的系统 I/O 和命令行输入模块都是高度依赖 UNIX 系统调用的，只能在遵循 UNIX 标准的机器上运行而无法在最流行的 Windows 平台使用。

6.2.3 不稳定的命令行输入模块

检查命令行参数的函数由于未知的原因不能同时接收两个以上的参数。鉴于 lac 解释器一共设计了三个命令行参数-g 、-help 和-std，因此-help 参数被设计成一经检测马上退出程序的选项。

另外，由系统保证的输入历史模块在 macOS 上表现也不稳定，经常会出现字符缓冲块不能完全被冲出的现象。

6.2.4 缺少宽字符支持

虽然 printf 函数族已经添加了对宽字符的支持，但语法捕获器本身不支持宽字符。另外，lac 系统的内存分配和转义字符处理也依赖基于标准 ASCII 字符宽度的长度计算，故要适配宽字符处理，需要对本系统作较大改动。

图 19 宽字符适配情况

6.2.5 import 文件路径逻辑

Lac 解释器的两种工作模式中使用了同一种路径逻辑，都以当前路径为“.”路径。这在执行不在当前文件夹的文件时，带路径的功能会产生歧义。

具体来说，lac 解释器在读取文件型输入时应该以当前文件路径为“.”路径，而在读取用户输入时以当前路径为“.”路径。添加相关特性需要在每个文件对象中存储其相应的地址，并依次更改当前路径的秩。

6.2.6 低效的内存分配设计

Lac 解释器表示数据结构的各类容器使用了数组作为了内存分配模型，且整个运行过程要频繁地创建和销毁容器对象。这在开发时导致了很多的堆内存污染错误。经过多次调试，这些错误大多数都被消除，但仍然留下了很多待优化的空间。

致谢

岁月不居，时节如流。大学时光匆匆而去，转眼间我又要启程向人生下一阶段进发。回首过去的四年时光，遗憾虽有，但更多的是收获，故在本文即将结束之际，我想对我生活和学习了四年的大学、指导我毕业论文的导师、帮助过我的老师和同学们致以诚挚的谢意。

首先我要感谢这所大学，它保留了旧时代高等学府最重要的特质：充足的课余时间和自由的课程安排。这让浸润在传统教育十八余载的我有机会实现思想启蒙，补上中学阶段缺失的文理教育课。在大学的四年里我阅读了大量人文社科类的经典作品，这些人类文明的隗宝重塑了我对周遭世界的感知和理解，赋予了我对未来生活的信心和力量。

然后我要感谢负责指导我完成毕业论文的老师。在本文的选题、构思和撰写过程中，章林忠老师给予了悉心指导和帮助，在论文的写作上章老师既给了我充分的自由空间，同时也花了很多心思帮助我修改论文，在此表示衷心感谢!

最后我要感谢帮助过我的老师和同学们，在大学四年中，我得到了很多同学和老师的热心帮助，也结下了很多深厚的友谊。衷心感谢与你们的相遇。

参考文献

[1] 周一萍,郑守淇,白英彩.基于 PVM 的并行 Lisp 机制与实现[J].小型微型计算机系统,1998(08):14-20.

[2] 王贺塞. 基于 Hindley-Milner 类型系统的函数式语言 Leaf 的设计实现[D].北京邮电大学, 2017.

[3] 黄文集. 形式规约语言 LFC 的实现和应用研究[D].中国科学院研究生院（软件研究所）, 2004.

[4] Yunmei Dong. Recursive functions of context free languages (II)[J]. Science in China Series : Information Sciences, 2002, 45(2) : 81-102.

[5] 马希文. Introduction to Theoretical Computer Science[M]. World Scientific Publishing, 1990.

[6] Lonsdorf B, Benkort M. Professor Frisby's mostly adequate guide to functional programming[J]. Source: https://mostly-adequate. gitbooks. io/mostly-adequate-guide, 2020, 1.

[7] 马希文, 宋柔. LISP 语言[M]. 高等教育出版社, 1990.

[8] Abelson H, Sussman G.J, Sussman J. Structure and Interpretation of Computer Program[M].The MIT Press, 2006.

[9] McCarthy J. Recursive functions of symbolic expressions and their computation by machine, part I[J]. Communications of the ACM, 1960, 3(4): 184-195., Part I.

[10] Felleisen M, Findler R B, Flatt M. Semantics engineering with PLT Redex[M]. Mit Press, 2009.

[11] Lippman S B, 侯捷. 深度探索 C++ 对象模型[J]. 2001.

[12] Aho A V. Compilers: principles, techniques and tools (for Anna University), 2/e[M]. Pearson Education India, 2003.

[13] Stevens W R, Narten T. UNIX network programming[J]. ACM SIGCOMM Computer Communication Review, 1990, 20(2): 8-9.

[14] Stevens W R, Rago S A, Ritchie D M. Advanced programming in the UNIX environment[M]. New York.: Addison-Wesley, 1992.

[15] Raymond E S. The art of Unix programming[M]. Addison-Wesley Professional, 2003.

Abstract

Design and Implementation of a Functional Programming Language Interpreter based on Lambda Calculus

Author: Gao Chengzhi Tutor: Zhang Linzhong
(School of Science, Anhui Agricultural University, Hefei 230036)

Abstract: In recent years, as major languages such as C, Java, Python, etc. have added support for functional programming pattern, this old-school technology is gaining grossly popularity again. Functional programming takes functions as its basic execution unit, which push the limit of the traditional procedural programming concepts and makes the ever complex software easier to be developed.

This paper designs a simple functional programming language Lachesis and its lac interpreter system based on lambda calculus, and it has been implemented using clang on a UNIX system. The interpreter could be divided into eight modules, namely environment management, built-in functions, container operations, parsing/lexical analysis, debugging, command line interaction, standard library and log modules, which is able to complete basic evaluation and variable binding, anonymous functions and other features, and has a considerable degree of scalability.

This paper first introduces the theoretical background of functional programming languages and its general characteristics before designs the core grammar of Lachesis language and the overall framework for implementing the lac interpreter system. Then the details of the lac interpreter development are illustrated. In the following chapters, the operating environment of the lac system and the test results of each unit are described. Finally, the work results during the subject are summarized with the further suggestions proposed.

Through the design and implementation of Lachesis programming language and lac interpreter, this paper introduces the core idea of functional programming and the general steps of implementing a new programming language interpreter in UNIX system.

Key words: computation theory; computer language; interpreter; UNIX programming;