基于 lambda 演算的函数式语言解释器的设计与实现

Contents

李京蔚，1234567890

浙江大学云峰学园

摘　要:

在信息化时代背景下，计算机操作系统在各行业领域中得到广泛应用。本文试图通过对计算机系统概念、功能和分类、历史的介绍，大致地给出对计算机操作系统的认识，并通过对当前主流桌面操作系统 Windows 和 macOS 的简要介绍，使读者能够对现代图形界面操作系统有一个较为全面的了解，从而喜欢上某一个操作系统，供读者参考。

关键词:

计算机；操作系统；功能；发展；Windows；macOS;

摘要

（**lambda** 演算是什么，有什么特点）

（本文立足于,,，理论，使用。。。开发,，什么效果）

（本文各章的安排）

序言

序言之序言

> 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。

本文要实现的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，是作者翻字典随便取的。

Lachesis 实现了一些函数式语言的最小功能集合，包括求值，变量绑定，函数、分支判断等等。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。

研究背景

~~20 世纪上半叶，对可计算性进行公式化表示的尝试有：~~

1. ~~美国数学家阿隆佐·邱奇创建了称为 演算的方法来定义函数。~~
2. ~~英国数学家阿兰·图灵创建了可对输入进行运算的理论机器模型，现在被称为通用图灵机。~~
3. ~~邱奇以及数学家斯蒂芬·科尔·克莱尼和逻辑学家 J. Barkley Rosser 一起定义了一类函数，这种函数的值可使用递归方法计算。~~

   ~~这三个理论在直觉上似乎是等价的□它们都定义了同一类函数。因此，计算机科学家和数学家们相信，可计算~~

论文的结构及研究内容

首先我们

Lachesis 解释器运用的计算理论

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 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 演算包括构造 lambda 项以及对它们执行的归约操作（reduction operations）。

Lambda 演算的项由以下三部分组成:

语法	名称	描述
x	变量	标识符名称，用来匹配函数中的某个名字
(λ x.M)	抽象	x 为变量名，M 为抽象，抽象默认采用波兰表达式。
MN	应用	抽象在参数前面，如 (λ x . plus x x) y => (plus y y)

或者用更简洁的巴科斯范式表示为:

$$
\begin{array}{lcll}
M, N, L & = & X & \\
 & | & (\lambda X.M) & \\
 & | & (MN) & \\
X & = & & \text{a variable: } x, y, ...
\end{array}
$$

Lambda 的规约操作则有两种:

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抽象（M）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 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。

Lachesis 解释器的需求分析和架构设计

Lachesis 语言设计

S-表达式

作为一门计算机语言，Lachesis 首先要能求值，例如:

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

事实上，鉴于绝大多数抽象语法树（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 + 不再是一个二元运算符，而是一个带有加法功能的符号。

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前缀表达式，也就是 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，这就是 Q-表达式，用一对花括号定义:

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

变量

我们可以通过 Q-表达式来定义变量的概念:

```
(_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赋值:

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

表达式也可以是变量，用 Q-表达式来传递形式，用 `eval` 函数将 Q-表达式转换成 S-表达式并推导。

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

lambda 函数

lambda 函数，或者叫抽象

实用函数

语言的细节是由解释器自身的结构所决定的

解释器设计

工作模式

现代解释器通常支持两种工作模式，一个是 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并马上返回结果。另一个则由用户事先写好脚本文件，解释器依次读取脚本并执行。

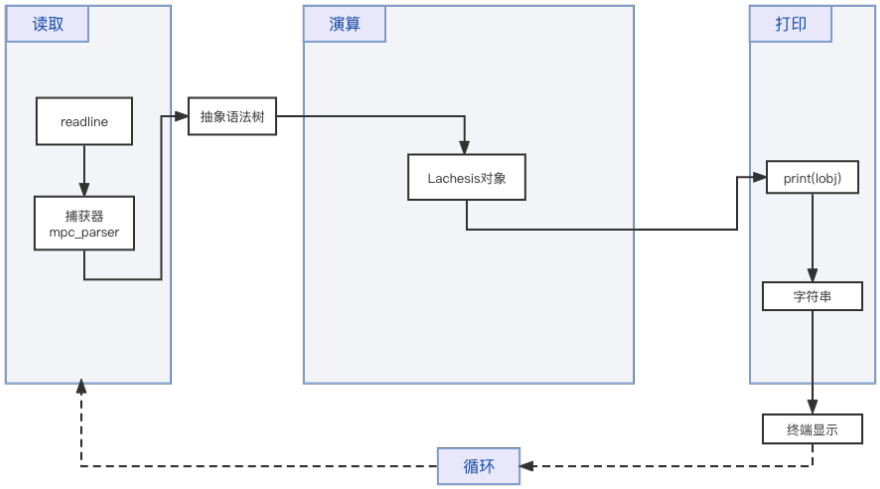

Figure 1: name1

（读取-求值-输出循环流程示意图）

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。

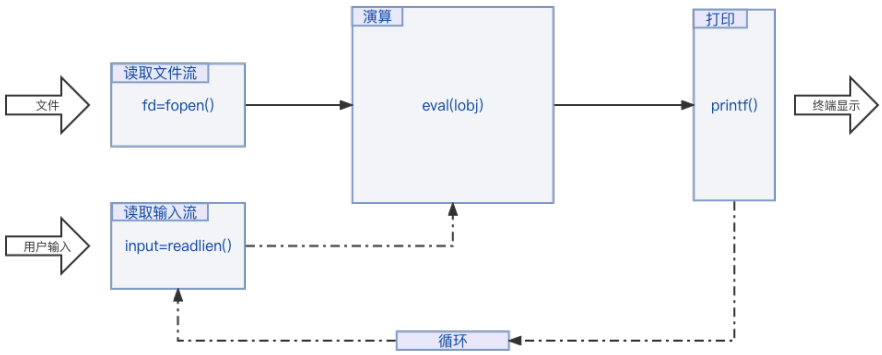

Figure 2: name12

演算模块的设计

我们要

Lachesis 解释器的具体实现

数据结构

在开发具有复杂数据结构操作的大中型程序时，最好设计一个公有基类作为容器，封装着信息在各函数的参数和返回值中传递。例如 macOS 和 iOS 的大部分对象都有一个基类 `NSObject`，子类继承了运行时系统的基本接口，以及作为 Objective-C 对象的能力。

Lachesis 在设计时借鉴了 Apple 的做法，我们用一个名为 `LObject` 的结构体封装了在程序中数据流动的所有类型，

```
LObject *lobj_number(long number);
LObject *lobj_error(char *fmt, ...);
LObject *lobj_symbol(char *symbol);
LObject *lobj_string(const char *string);
LObject *lobj_sexpr(void);
LObject *lobj_qexpr(void);
```

.....

并用一个枚举类型来识别该结构体的属性，如果这个 `LObject` 结构体的类型是 S-表达式或者 Q-表达式，那么它则会存储了指向其他 `LObject` 的指针以及指针的个数，方便将来作递归式的推导。

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抽象语法树的过程。

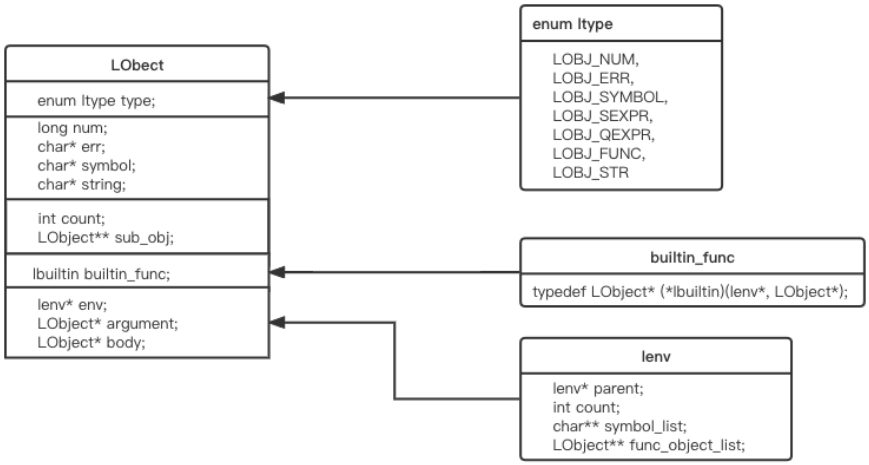

Figure 3: name3

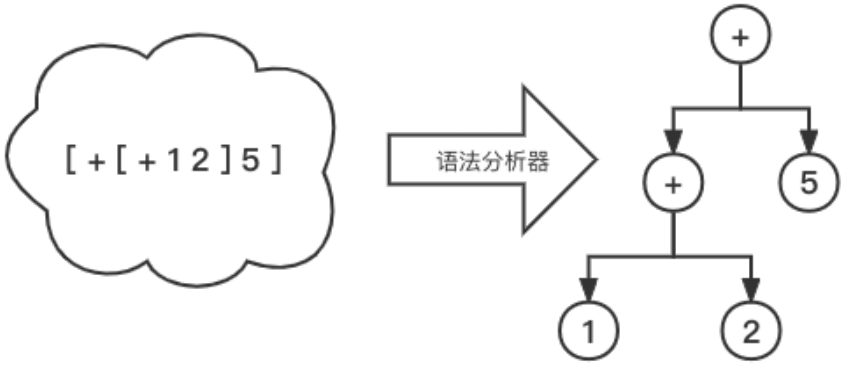

Figure 4: name4

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。

```
mpca_lang(MPCA_LANG_DEFAULT, "                                  \
    number    : /-?[0-9]+/ ;                                    \
    symbol    : /[a-zA-Z0-9_+\\-*\\/\\\\=<>!&]+/;               \
    string    : /\"(\\\\.|[^\"])*\"/;                           \
    comment   : /;[^\\r\\n]*/;                                  \
    sexpr     : '[' <expr>* ']' ;                               \
    qexpr     : '{' <expr>* '}' ;                               \
    expr      : <number>|<symbol>|<string>|<comment>            \
              | <sexpr> | <qexpr> ;                             \
    lexpression     : /^/ <expr>* /$/ ;                         \
    ",
    Number, Symbol, String, Comment, Sexpr, Qexpr, Expr,
    LExpression); // notice, can't place space after '\' !
```

可以看出,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 raw 上。

```
if (mpc_parse("<stdin>", input, LExpression, &raw)) {……}
```

抽象语法树的推导

语法分析器捕获的结果是一个多叉树。

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

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。

举个例子:

```
Lachesis> + 13 [ - 15 12 ]
Sucess to build the AST tree!
>
  regex
  operator|char:1:1 '+'
  expr|number|regex:1:3 '13'
  expr|>
    char:1:6 '['
    operator|char:1:8 '-'
    expr|number|regex:1:10 '15'
    expr|number|regex:1:13 '12'
    char:1:16 ']'
  regex
Lachesis>
```

我们要读取抽象语法树中的内容并建立相应的 LObject

环境与变量

我们希望

内建函数设计

动态调试器模式和错误处理

在 Lachesis 的

标准库设计

从匿名函数到实名函数

我们内置的 lambda 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}]
```

系统结果分析

测试用例

总结与展望

开发时遇到的一些技术细节

尽量不要使用宏替换

目前的智能语义重构器不支持对宏的扫描

Escape string 与 Unescape string

致谢

参考文献

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

66

附录