TinyScheme——一个微型 Scheme 语言解释器

柯嵩宇
上海交通大学

2015 年 8 月 17 日

Contents

1 简介

一个实现了基本 Scheme 语言语法的解释器。

2 运行环境

解释器用 Javascript 编写，运行环境为 node.js(v0.12.7)。使用了开源的高精度库 number-crunch 实现对大整数运算的支持。项目目录中已经包含了该模块 (./node_modules/number-crunch)，若没有该模块可以执行：`npm install number-crunch` 获取。[1]

执行的时候保证 main.js,util.js 以及包含 number-crunch 库的 node_modules 目录在一个文件夹下，在终端中执行命令“node ./main.js”。解释器会从”./src.scm” 中读取 Scheme 并解释执行，同时将执行结果输出到标准输出 (stdout) 上。

2.1 number-crunch 库简介

用 Javascript 编写的开源高精度库，用 28 位整数数组来实现对大整数的存储和运算，提供了将字符串表示的大整数转换为运算所需的整数数组以及将结果重新转换成字符串表示的函数。

支持的大整数操作包括：

- 加、减、乘、除、取模
- 大整数开平方根（向零取整）
- 乘法逆元等常用的整数运算算法。

3 项目的实现

3.1 运行细节

解释器启动时先进行一系列的初始化操作后会从”./src.scm” 中读取 Scheme 语言语句，然后通过 util.GetSententes 函数将源代码文件分割为单独的语句并且逐句执行。

通过 ProcExec 函数执行已划分好的每一条语句。

ProcExec 通过调用 util.GetElements 进行语句的分词，获取每一个参数的类型，然后通过 ProcessParas 处理参数，获得参数的实际值，然后根据被调用的过程来执行不同的处理操作。

3.2 语法和函数

在编写代码的时候，我把过程分为语法 (syntax) 和函数 (function) 两类，两者的区别是，函数有自己的域 (scope)，管理函数内定义的局部变量，而语法没有，执行语法时始

[1]通过 npm 方式获取的 number-crunch 库有一个小 bug——不能够将内部数据类型的负数转换成字符串形式，我在 GitHub 上与源代码一起提供了我修改后的 number-crunch 库。

终从当前可见域中获取数据。由于 Javascript 对于对象的赋值都是地址赋值，所以在传递参数的时候加上可见域不会对效率造成致命的影响。[2]

在 Scheme 语法标准中，Scheme 的解释器必须对尾递归进行优化，但是由于我不能找到正确判断是否为尾递归的方法，因此就没有对函数尾递归进行优化。

3.3 数据的存储

Scheme 语言本身是弱类型的，但是在运算时不同的数据类型会有不一样的表现。同时考虑到 Javascript 同样也是弱类型语言，解释器采用了下面的 object 来保存数据：

```
{
    "type" : ***,
    ***
}
```

3.4 支持的数据类型

1. integer
2. float
3. char
4. string
5. pair
6. list
7. symbol
8. function

3.5 函数的实现细节

采用了下面的结构来保存函数的信息：

```
{
    "type" : "function",
    "paraList" : [...],
    "body" : [...],
    "scope" : {...},
}
```

type 表示这个对象的类型是一个函数。

paraList 函数的形参列表。

[2]函数定义时会记录当前的可见域，同时在函数调用的时候会对该函数的可见域进行一层拷贝（不需要深度拷贝），因此，在大量递归的时的表现并不是很好，而且由于 Javascript 栈空间不足，N 皇后问题最多只能支持到 7 皇后。

body 保存函数体的每一条语句。

scope 保存函数执行前的可见域。[3]

函数在执行时会对 scope 执行一次复制（浅复制），然后以副本作为当前的可见域执行语句。

3.6 一些等价的转换

3.6.1 let

在实现中，把 let 环境转换为 lambda 表达式执行，如：

```
(let ((var1 expr1)
      (var2 expr2)
      ...
      (varN exprN))
     body1
     ...
     bodyN)
```

转换为：

```
((lambda (var1 var2 ... varN)
    body1
    ...
    bodyN)
    expr1 expr2 ... exprN)
```

3.6.2 let*

在实现时，let* 环境被视为 let 环境的嵌套，如：

```
(let* ((var1 expr1)
       (var2 expr2))
      body1
      body2)
```

转换为：

```
(let ((var1 expr1))
    (let ((var2 expr2))
        body1
        body2))
```

3.6.3 letrec

letrec 环境被 lambda 语句重新封装，如：

```
(letrec ((<var1> <expr1>)
         (<var2> <expr2>))
        body1
        body2)
```

[3]用于实现 Lexcial Scope

转换为：

```
((lambda ()
    (define <var1>.1 expr1)
    (define <var2>.1 expr2)
    (define <var1> <var1>.1)
    (define <var2> <var2>.1)
    body1
    body2))
```

4 实现的语法

4.1 基本语法

- define
- lambda
- map
- apply
- if
- cond
- begin
- quote
- display
- newline
- let,let*,letrec

4.2 数值运算相关

- $+, -, *, /$
- quotient
- modulo
- sqr

4.3 逻辑运算相关

- and
- or
- not

4.4 list 和 pair 相关

- list
- cons
- car
- cdr
- length
- append
- reverse
- list-ref
- memq
- assq

4.5 一元断言与二元断言

- eq?[4]
- eqv?
- equal?
- $<, >, =, \leq$
- zero?
- string=?
- null?
- pair?
- list?

5 总结

5.1 过程中的收获

开始的时候由于考虑用什么语言来完成纠结了很久。先后考虑了 C++、Python、Haskell 和 Javascript，但是由于我之前并不会这两门语言，所以用了一个星期的时间来学习。最后基于以下几点选择了 Javascript：

[4]该断言可能存在一些问题，在 MIT Scheme 的参考手册中，eq? 的存在大量未定义的行为。

1. Scheme 是弱类型语言，而 Javascript 也是弱类型语言，这样可以避免用强类型语言来模拟弱类型语言时复杂的抽象过程。

2. Javascript 的执行效率令人满意，不像 Python 的效率那么令人堪忧。

3. Javascript 的语法与 C++ 相似。

4. 可以轻松的获得开源的高精度运算库。

在完成解释器，积累了大量的经验之余掌握了两门编程语言，这也算是额外的收获吧。

5.2 设计上的优点

本人对自己的设计还是比较看好的。利用 Javascript 的语言特性使得我可以很轻松地在现有代码的基础上增加对新的语法的支持而不需要改动大量的代码。[5]

5.3 关于语法糖的一些看法

由于 Scheme 本身并不支持迭代，需要通过递归操作完成对 list 的遍历。在没有对尾递归进行优化的情况下会造成递归引起巨大的内存消耗，从而让程序的执行效率降低。因此，虽然在编写中使用“语法糖”可以减少很多的代码量，但是这会使得许多原本可以用线性迭代模拟的操作因为没有尾递归优化而变成了线性递归的操作，这是让人不能接受的。[6]

5.4 存在的一些问题

由于 Javascript 的栈空间不足，在大量深度递归的时候有可能会栈溢出。同时由于代码在处理函数递归时的优化较少，出现复杂的函数调用时效率会受到一定的影响。同时变量内存空间的管理完全地托管给 Javascript 的垃圾回收机制故不适合处理大量数据的情况。

[5] 只要在./main.js 中加入对相关语法行为的定义就好了。

[6] 虽然我不支持使用它，但是在实现 let 环境的时候我还是“偷懒”地用了，因为把它等价转换之后实在是太简单了。