元循环求值器大作业报告

1200012741 史杨勋惟

简介：

相比于其他三个作业，这个作业和书上课上的内容相对比较接近，我选择这个一方面是为了对课上的内容加以巩固和理解，另一方面也是想更深入的了解一下一个程序语言的设计流程。

主要完成工作：

一共两个文件，beval.scm 是书上的元语言求值器的一个简单扩展，banal.scm 是对应的语法分析器。主要完成的扩展内容基本和要求一致，定义完备，在 racket 下的 r5rs 中可以正常运行，，按照指定要求添加了 and，or，let 功能，完善了内部定义，添加了 map，while 以及计时功能。并做了两个程序的对比试验。

设计流程：

收集书上关于元语言求值器以及语法分析器的一些代码，然后根据相关的习题进行了一些扩展。

首先是让程序正常运行起来，添加了 true 和 false 的定义，还有 error 的定义。关于 error 的定义，我在网上查阅了一下，scheme 中 error 的定义最后都有一句 (scheme-get-environment -1)， 这一句话是用来传递异常的貌似，我把这句话删了，因为我觉得这样的话可以不退出，在报错了以后依然保持用户的环境，这样对用户比较友好。

接下来是基本的扩展，包括 and, or, let 以及基本的算术运算。这些内容还是比较简单的，因为 let 很容易转换程基本过程，而 and 和 or 的扩展也没有难度。

然后是关于 map 的问题，把 scheme 内部的 map 拿出来是错误的，因为 scheme 内部的 map 会把一个基本过程作为参数，但是，在这个求值器中，基本过程是以一种数据结构的形式存在的，有标签有环境，所以直接使用 scheme 内部的操作会出错。我添加了 map 功能，但是添加的基本方法是把 map 拆成一个函数的定义和执行的两个过程，其实归根结底和直接在求值器环境下定义 map 本质上是一样的。

最后是 while，和 map 异曲同工，也是一个定义加一个执行。

完成了求值器后，我就选择了完成语法分析器。这里只要简单的把书上的代码收集了一下就可以正常工作了。其实起初我还是不太理解为什么这样做可以大大节省时间，我一开始认为既然分析的内容是无法存在环境中的，那么每次求值调用的过程还是要分析很多次啊~ 于是我就开始边调试边分析，后来发现其实这个语法分析的本质和编译很相似。如果我们把求值器内部定义的过程称为“内部过程”，在 driver-loop 外部定义的过程，比如一些求值器内核过程称为“外部过程”，那么 analyze 就是一个把过程从“内部过程”翻译到“外部过程”的过程，而外部过程就，即我们熟悉的 scheme 过程又是通过 let 的方式直接保存在了局部环境中，所以每次求值就可以直接调用而不需要在此翻译了。

随后我为两个程序添加了计时功能，由于 r5rs 的库中我没有查到有计时功能，所以我的选择是使用了 mit-scheme 中的 get-universal-time，所以**如果要使用计时功能，需要在 mit-scheme 中运行此程序。**

实验

我选择了四个实验：

实验 1：树形 fib

树形 fib 的执行次数应该是和所求出的斐波那契数是成正比的，而两个程序的运行时间也有差异，(fib 22)在 beval.scm 中运行的时间是 15s，在 banal.scm 中运行的时间是 7s。

实验 2：线性 fib

线性 fib 的效率远大于树形 fib，所以我们需要较大参数，实验结果为(fib 100000)在 beval.scm 中的运行时间位 9s，在 banal.scm 中的运行时间位 4s

实验 3：汉诺塔

汉诺塔的源程序中有输出操作，这个会占用很大时间，所以我把所有的输出都删去了，实验结果为 15 个托盘的汉诺塔程序在 beval.scm 的运行时间为 14s，在 banal.scm 的运行时间为 6s。

实验 4：合并有序表

合并有序表是归并排序的基本操作，时间复杂度也为线性，我手动生成了两个大约 10000 个数的有序表，测试情况为 beval.scm 5s，banal.scm 2s

综上所述，语法分析器的执行效率约为求值器的 2 倍左右。当然，上述过程的特点是同一个过程均会被执行很多次，所以当我们调用非递归的普通过程的时候，这一个差距会变得很小，甚至有时 beval.scm 会超过 banal.scm，因为它可以不经过分析直接求值。

缺陷与展望：

有一个比较遗憾的缺陷是，我曾经想过给这个语言求值器增加调试功能以及一些错误恢复功能，但最终没有实现。错误恢复需要我们记忆下每步进入 eval 的过程，这个也许需要全局数据结构来进行维护，不过我遇到的问题在于反执行这个过程有以下困难，错误步骤不执行是可以的，但是如果要停在程序执行的中间阶段这个也不好实现。关于调试，这个我也想过，而且 scheme 里也有这个功能，但是我具体想了一下，觉得这个内容还是有一些复杂，可能我们需要在求值器外另外写一个调试器来完成一些基本的功能，可能可以通过流来实现，但是由于时间原因，在尝试了一会儿后我还是选择了放弃。

总结：

这次大作业让我对语言这个概念有了更深入的认识，尤其是对 scheme 或者 lisp 的认识。在 lisp 中一切都是表，lisp 不仅有一套很好的处理表结构的机制，其实他的语言本身也是通过表结构实现的，所以这样就可以轻松地通过 lisp 来设计 lisp 了。

当然，要设计一个好的程序设计语言光靠这些还是不够的，我们还需要添加更多的功能，而这些功能可能还需要对计算机更进一步的认识，而且执行效率也是我们所需要强烈关注的。虽然还有很多改进空间，但这次大作业也已经着实开阔了我的眼界，让我有了一些更深的见解。