Design and Implementation of SJTU-Schemer
a Purely Functional Approach

Jingcheng Liu （刘景铖）*

August 8, 2011

1 Introduction

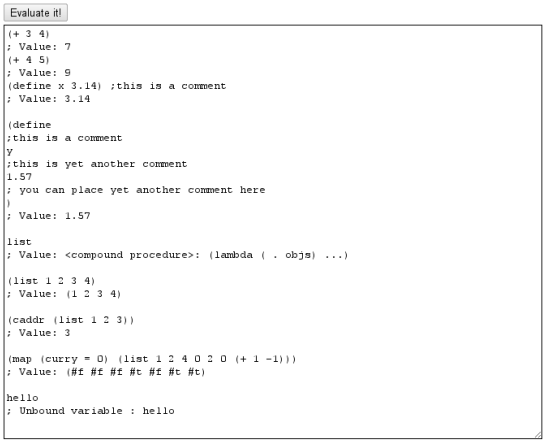
1.1 Usage

要使用命令行下面的解释器，请在 schemer 根目录下面执行 make. test.scm 中包含了一些可供测试用的代码.

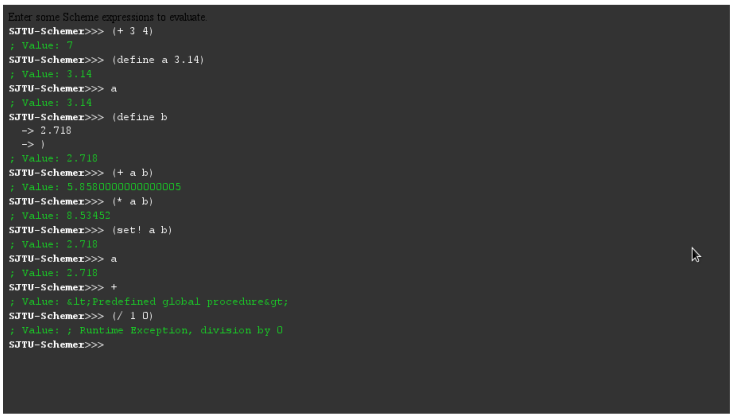
要使用在线版解释器，请在 schemer/web 目录下面执行`cabal install`. 接下来请在相同目录下面执行`web -p 8000`. 之后可以访问`http://localhost:8000/parser/` 和`http://localhost:8000/schemer/` 查看不同实现效果.

1.2 Screenshot

SJTU-Schemer 0.1 Alpha by Liuexp.



SJTU-Schemer 0.1 Alpha by Liuexp.



*F1003028(ACM Honored Class) 5100309243 `Liuexp@gmail.com`

```
File  Edit  View  Search  Terminal  Help
[liuexp@liuexp haskell]$ ./main
Welcome to SJTU-Scheme!  Type: (quit) to quit.
stdlib loading...[DONE]

SJTU-Scheme>>> (define (test a)
... 1 more >(define b 1)
... 1 more >(display b)
... 1 more >(set! b a)
... 1 more >(display b)
... 1 more >(+ 3 4)
... 1 more >)
;Value: <compound procedure>: (lambda ("a") ...)
SJTU-Scheme>>> (test 2)
1
2
7
SJTU-Scheme>>> (test 3)
1
3
7
SJTU-Scheme>>> (test 9.9)
1
9.9
7
SJTU-Scheme>>> (test 2+3i)
1
2.0+3.0i
7
SJTU-Scheme>>> (test 2-3i)
1
2.0-3.0i
7
SJTU-Scheme>>> (test (/ 2 3))
1
2/3
7
```

```
File  Edit  View  Search  Terminal  Help
SJTU-Scheme>>> (filter (lambda (x) (> x 0)) '(1 2 3 -4 2/5 -1.1 3.4 -2/5))
(1 2 3 2/5 3.4)
SJTU-Scheme>>> (filter (lambda (x) (> x 0)) '(1 2 3 -4 2/5 -1.1 3.4 -2/5 2+3i -2+3i))
Invalid type: expected orderable number, found (-2.0+3.0i 0)
SJTU-Scheme>>> (filter (lambda (x) (> x 0)) '(1 2 3 -4 2/5 -1.1 3.4 -2/5 2+0i))
Invalid type: expected orderable number, found (2.0+0.0i 0)
SJTU-Scheme>>> length
;Value: <compound procedure>: (lambda ("lst") ...)
SJTU-Scheme>>> (length '(1 2 3 4 5 6 7))
7
SJTU-Scheme>>> (max 7 4 5 6 2 5 9 1 55/3 7/8 23.1)
23.1
SJTU-Scheme>>> (map (lambda (x) (> x 0)) '(1 2 3 -4 2/5 -1.1 3.4 -2/5))
(#t #t #t #f #t #f #t #f)
SJTU-Scheme>>> (map (curry < 0) '(1 2 3 -4 2/5 -1.1 3.4 -2/5))
(#t #t #t #f #t #f #t #f)
SJTU-Scheme>>> (and #t #t #t #f #t #f #t #f)
#f
SJTU-Scheme>>> ((lambda x x) 3 4 5 6)
(3 4 5 6)
SJTU-Scheme>>> ((lambda (x y . z) z)
... 1 more >3 4 5 6)
(5 6)
SJTU-Scheme>>> (equal? "abc" "abc")
#t
SJTU-Scheme>>> (eqv? (cons 1 2) (cons 1 2))
#t
SJTU-Scheme>>> (max 7 4 5 6 2 5 9 1 55/3 7/8 23.1 25
... 1 more >)
25
SJTU-Scheme>>> (not 3)
#f
SJTU-Scheme>>> (not #t)
#f
SJTU-Scheme>>> (not nil)
#f
```

```
SJTU-Scheme>>> (+ #o10 #xa #xA #b10)
30
SJTU-Scheme>>> (define (x y z) (lambda (m) (m y z)))
;Value: <compound procedure>: (lambda ("y" "z") ...)
SJTU-Scheme>>> (define (w z) (z (lambda (y z) y)00
... 2 more >))
;Value: <compound procedure>: (lambda ("z") ...)
SJTU-Scheme>>> (define (w z) (z (lambda (y z) y)))
;Value: <compound procedure>: (lambda ("z") ...)
SJTU-Scheme>>> (w (x 1 2))
1
SJTU-Scheme>>> (define (ww z) (z (lambda (y z) z))
... 1 more >)
;Value: <compound procedure>: (lambda ("z") ...)
SJTU-Scheme>>> (ww (x 1 2))
2
SJTU-Scheme>>> ((x 1 2) +)
3
SJTU-Scheme>>> ((x 1 2) -)
-1
SJTU-Scheme>>> ((x 1 2) *)
2
SJTU-Scheme>>> ((x 1 2) /)
1/2
SJTU-Scheme>>> (define (a b c)
... 1 more >            (= b (+ (* c
... 4 more >                            (quotient b c)
... 4 more >                    ) (remainder b c) ) )
... 1 more >)
;Value: <compound procedure>: (lambda ("b" "c") ...)
SJTU-Scheme>>> (a 1 2 )
#t
SJTU-Scheme>>> (a 3 4)
#t
SJTU-Scheme>>> (a 1013 1600)
#t
```

```
File  Edit  View  Search  Terminal  Help
SJTU-Scheme>>> (fac
... 1 more >1
... 1 more >)
1
SJTU-Scheme>>> (+ 1 2 3 4 5 6 7 8 9 10)
55
SJTU-Scheme>>> (- 1)
-1
SJTU-Scheme>>> ''''a
(quote (quote (quote a)))
SJTU-Scheme>>> (car ''a)
quote
SJTU-Scheme>>> (cadr ''a)
a
SJTU-Scheme>>> (cadddr ''a)
Invalid type: expected pair, found ()
SJTU-Scheme>>> (car nil)
Invalid type: expected pair, found ()
SJTU-Scheme>>> (display (car ''a))
quote

SJTU-Scheme>>> (< 1 2 3 4 5 6 7 8 9.0 101/9)
#t
SJTU-Scheme>>> (< 1 2 3 4 5 6 7 8 9.0 101/17)
#f
SJTU-Scheme>>> (< 1 2 3 4 5 6 7 8 9.0 101/97 3+4i)
Invalid type: expected orderable number, found (1 2 3 4 5 6 7 8 9.0 101/97 3.0+4.0i)
SJTU-Scheme>>> (< 1 2 3 4 5 6 7 8 9.0 101/9 "abc")
Invalid type: expected complex number, found ;Value: "abc"
SJTU-Scheme>>> (= 0 0.0 0/1 0+0i)
#t
SJTU-Scheme>>> (symbol? 1)
#f
SJTU-Scheme>>> (symbol? a)
; Unbound variable: : a
SJTU-Scheme>>> (symbol? 'a)
```

2 Basic Framework

2.1 Monad

由于本文介绍的解释器是基于 Haskell 语言实现的, 这里简单介绍一下里面的 Monad.

首先，对于纯函数式代码来说，一切行为是完全确定的，也就无法直接和现实世界交互，因为现实世界的行为和参数都是不确定的。这就需要一些介质来与现实世界交互了,Haskell 里面就是通过 Monad 来实现的，也

就是通过 Monad 来把纯函数式代码与那些将直接与现实世界交互的代码分离开来。

Monad 的出现主要是为了简化一些计算步骤的组合过程的. 就像 Java 中的 try..catch..finally 一样，不过不同的是 Monad 要比它更简洁, 不同的 Monad 实际上是对计算过程的不同组合方式.

比如说 Maybe Monad 是用于那些可能成功（则返回结果），也可能失败（则什么也不返回）的过程的组合，就像命令式语言中我们会返回 0，-1，INT_MAX 一样，不过这些值本身是没有意义的, Maybe Monad 里面则有了明确意义. 而 MonadPlus 在 Maybe Monad 下面的组合将表现为如果有一个 Maybe Monad 里面成功了则返回里面的值，否则什么也不返回（注意这也是在一个新的 Maybe Monad 下面）. 这些过程都很常见，把它们的过程特征抽象成 Monad 只是极大的简化了代码.

而有趣的事情是，这些计算过程什么时候能抽象成 Monad，是需要满足一些条件的，这些条件和范畴论中的 Monad 所要满足的条件是完全一样的，这也是 Haskell 中会出现范畴论中的词语的原因.

2.2 Parsing

Scheme 的语法结构比较简单，可以分两种，一种在 Lisp 中被称为原子 (**Atom**)，意即不可再分的东西，如一般的数字，字符串，符号，布尔值都属于原子；另一种则是表达式，表达式用括号括起来. 另外空格是作为分隔符.

总的来说 Parser 还是比较好写的，不过也比较容易在一些细节上面出现 Bug, 比如表达式前面的空格，括号前后的空格等.

在引入注释之前可以直接把换行符全部替换成空格, 这样就可以处理多行的问题了. 引入注释后就不能这么处理了, 一种方法是先处理掉注释，不过这里采用的方法是在 Parser 中同步处理注释.

编写 Parser 时，一种方法是自己写，主要是实现两个过程, 一个是 peek 即向前看的, 一个是 eat 就是处理掉一个字符的. 不过自己编写时如果没有做好模块化的话，后面修改起来会非常困难，而且因为抽象层次不高，即使有 Bug 也只能在测试中发现. 另一种方法是直接使用已有的工具生成, 如 C++ 的可以用 Yacc，Java 的可以用 JavaCC，这些工具可以直接通过 BNF 文法生成需要的 Parser. 还有一种方法是使用一种既可以像 BNF 文法那样编写 Parser, 还有一些现成的 Parser 来自由组合的 parser combinator library, 如 Haskell 中的 Parsec,Java 中的 JParsec 等. 这里用的是 Haskell 中的 Parsec.

下面是一段利用 Parsec 来处理 list 一类的表达式的示例，它是在 Parser 这一 Monad 中进行的，返回值是 Scheme Primitives 类型. 其中`char '('` 即指让 Parser 吃掉这个字符. 而`try ...` 是指让 Parser 向前看看是不是有这么一些字符.

```haskell
listParser :: Parser Primitives
listParser = do
        char '('
        try (skipMany space)
        x <- sepEndBy exprParser spaces
        try (skipMany space)
        char ')'
        return $List $ filter (not . isComment) x
```

下面一段则是处理复数类“原子”的一个 Parser. 这些过程中会有很多细节, 比如复数并不总是 $1+2i$, 可能是 $1-2i, -1+2i$ 之类的. 而在显示的时候, 如果虚部是负数那么就不需要加符号直接显示, 如果是正数或 0 则需要加一个“+”后再显示等.

```haskell
complexParser :: Parser Primitives
complexParser = do x <- (try realParser <|> numberParser)
                   z <- try (char '+') <|> char '-'
                   y <- (try realParser <|> numberParser)
                   char 'i'
                   return $ Complex ( toReal x  :+ case z of
                                                     '+' -> toReal y
```

```
                                                    '-' -> (0 -). toReal $y
                                                    )
```

2.3 Evaluation

这一部分对于使用 Python 的同学可能会比较容易实现, 因为 Python 本来就有一个 eval 的函数. 用 Python 写的 Scheme 解释器如 pyscheme 等都采用了内置的 eval 函数.

如果按照纯函数式代码与需要与外界交互的代码分离的原则, 那么这里的 eval 是应该属于纯函数式代码的. 不过仔细一想却发现并不是如此的, 首先 Scheme 本身并不是一门纯函数式编程语言, 所以 eval 里面会涉及赋值等改变环境, 以及类似`(display "sth")` 这样的直接的 IO 操作.

看起来把 display 当成直接返回会是一个解决方法，不过这却是错误的，因为对于多条语句同时执行的过程中, eval 只能返回最后一条的执行结果, 如果 display 正好处于中间那就被跳过了.

所以无论如何，对于 Scheme 解释器,eval 都是需要放进 IO Monad 的. 所以这里对于环境选择的也是 IORef 而不是 STRef.

另外在 Haskell 中引入完整的数值系统时可能会遇到一些麻烦. 因为在像 Haskell 这么强的类型系统中，要想像 C++，Java，Python 中那么随便地进行通用型数值运算是不可能的，每个过程都必须有其确定的行为，如取模接受的只能是整数，大于号是不接受复数的等等，在这种情况下面，要想给环境 bind 上一个通用的 +,-, 取模等运算符是不可能的（因为参数和返回值的类型不一致），相对来说，比较诱人的但是却是错误的做法则是在运算 (eval) 的过程中把 +,-等也作为一种 Special Form，因为这时能拿到运算时的参数列表，可以通过参数的类型来确定 +,-等所需要的实例的计算过程了。不过这样 +,-就不能作为一个计算过程进行传递的了。我最后想到的比较简单的方法是环境中绑定的并不是真的 +,-计算过程，而只是一个虚的接口，这样在这个虚的接口里面也只用在运算前先对参数作一个类型检查，然后就可以通过类型来确定 +,-所需要的实例的计算过程了。

2.4 Apply

这一部分对于使用 Python 的同学可能会比较容易实现, 因为 Python 本来就有一个 apply 的函数. 用 Python 写的 Scheme 解释器如 pyscheme 等都采用了内置的 apply 函数.

Eval 和 Apply 的时候主要需要注意 closure 的问题. 下面用一个例子说明这里采用的解决方法.

2.5 Closure

不妨来看下面的代码.

```
(define (test x)
        (define y 1)
        (display y)
        (set! y x)
        (set! pi x)
        (display y)
 )
```

那么在全局中会绑定 test 这个 symbol 到一个 procedure 上面. 不过 procedure 会比较特别一点，因为需要一个 closure 的东西来确保在内部可以访问并修改外部，但在内部新定义的东西却又不会影响外部. 所以在绑定到 procedure 的同时还会把定义这个 procedure 所在环境的引用绑定上去. 当调用它, 如执行`(test 9)` 时，就要在全局这个环境中找到 test 对应的这个过程, 然后执行它，执行的时候会为 test 找到所在的 closure，然后在里面给 x 绑定上 9，`(define y 1)` 时就在这个独立于全局的 closure 中给 y 一个值,`(set! y x)` 并修改它为 9，最后结束后全局的环境中 y 还是未定义的, 但 pi 的值会被修改为 9.

3 Online Interpreter

在做好一个在命令行下面的后，可以开始想怎么做一个在线的，在线的本身并不是重点，重点在于如果有了一个在线的模型后，就有了一个通用的 API 接口可以让它用于其它用途，如手机等各种嵌入式的本质上也只是把运行的平台换一换罢了，接口基本上不需要什么变化。

不过如果是在 Haskell 下面的话, 这种担心是多余的，Haskell 语言设计中的机制——把纯函数式代码与需要和外界交互的代码分离，其中后者是被包装到了 Monad 中——这是借用自范畴论的思想，这也使得最后转换为在线版的过程非常的顺利——仅仅是加了一个 Monad Transformer——把原来的 IO Monad 中的 action 就转换到了 Snap.Extension Monad 中了（一个 Web 服务器框架），原来的代码甚至不用作任何修改。当然，这并不是真的不用修改，在线版与本地命令行的很重要的区别是 Web 的安全问题，尤其像这样的解释器，如果直接基于源代码进行过滤毕竟是有安全隐患的，所以我采用的是直接把对应的如打开文件，修改文件等命令去掉。这个 API 接口倒是通用的了。

另外在线的解释器需要考虑的一个问题是, 如何保存用户之前的环境. 在本地中我们可以采用 curry 的方式, 在 REPL 的过程中一直调用这个 partially applied function 来实现使用的一直是同一个环境.(这一行代码太长故只保留了最核心的 Monad Binding)

```
1 primitiveBindings >>= evalAndPrint
```

但在 Web 下面就不可以了, 因为也没有 REPL 这个循环，用户也不是单一的. 最简单的方法自然就是把代码保存下来. 这也是这里采用的方法.

理论上是可以通过类似 Http Session 的方法来解决的, 又或者是通过数据库（但数据库如何保存一个 procedure 我还没想到简单一点的解决方法，haskell 中的数据库似乎也只有那些一般 SQL 数据库中的原始的数据类型)

考虑我对这方面了解不多，所以这里实现了的主要还是两方面的内容, 一是类似 MIT-Scheme 通过 edwin 嵌入到 emacs 编辑器的，通过一个编辑窗口，允许用户在编辑的过程中，随时 evaluate。另一个是受到 tryhaskell 的启发，模仿本地的界面做出来的一个在线解释器.

4 Further Reference

- Java 的可以参考 Jscheme.
- Python 的可以参考 pyscheme.
- Haskell 的可以参考 Write Yourself a Scheme in 48 Hours.