1 介绍

本次作业中, 你需要实现一个计算器. 我们先介绍一下在这个计算器中, 我们如何写算式.

```
Scheme] 1
   1
Scheme] (+ 1 1)
   2
Scheme] (* (+ 1 1) (- 2 1))
   2
```

除了数字, 我们还有布尔值和 `if` 表达式. 注意, 事实上除了 `#f`, 其他值均被认为是真.

```
Scheme] #t
   #t
Scheme] #f
   #f
Scheme] (if #f 0 1)
   1
Scheme] (if 1 #t #f)
   #t
```

除了布尔值, 我们还有一个重要的类型: 对.

```
Scheme] (cons 1 2)
   (1 . 2)
Scheme] (car (cons 0 1))
   0
Scheme] (cdr (cons 1 (cons 3 4)))
   (3 . 4)
```

一个常见的操作是将一系列对连起来, 组成一个列表. 打印列表时不会出现上面的句点, 正如我们写算式一样. 一个空列表可以用 `(quote ())` 表示.

```
Scheme] (quote ())
   ()
Scheme] (cons 1 (cons 2 (cons 3 (quote ()))))
   (1 2 3)
Scheme] (cons 1 (cons 2 (cons 3 4)))
   (1 2 3 . 4)
```

空列表用 `(quote ())` 表示的逻辑是, 我们用 `quote` 将一个算式变成这个写这个算式的列表, 正如这个苹果和"这个苹果".

```
Scheme] (quote (1 2 3 4))
   (1 2 3 4)
Scheme] (cdr (quote (1 2 3 4)))
   (2 3 4)
```

只有数字太无聊了, 幸好 `quote` 暗示了我们可以得到类似枚举类型的符号.

```
Scheme] (quote me)
   me
```

有一系列判断数据类型的谓词.

```
Scheme] (fixnum? 1)
   #t
Scheme] (boolean? #t)
   #t
Scheme] (null? (quote ()))
   #t
Scheme] (null? 3)
   #f
Scheme] (pair? (quote ()))
   #f
Scheme] (pair? (quote (1)))
   #t
Scheme] (symbol? (quote a))
   #t
```

我们可以给一个算过的表达式名字. 注意这和符号的区别.

```
Scheme] (let ((2^1 2))
          (let ((2^2 (* 2^1 2^1)))
            (let ((2^4 (* 2^2 2^2)))
              (* 2^4 2^4))))
   256
Scheme] (let ((x (quote x)))
          (let ((x (cons x (quote y))))
            x))
   (x . y)
```

一般的计算器就到此为止了. 然而, 我们的计算器甚至是可编程的. 我们可以定义函数.

```
Scheme] (lambda (x) x)
   #<procedure #f (x)>
Scheme] ((lambda (x) x) (lambda (x) x))
   #<procedure #f (x)>
Scheme] (let ((square (lambda (x) (* x x))))
          (square (square (square 2))))
   256
Scheme] (procedure? (lambda () (quote ())))
   #t
```

为了方便地写递归函数, 有一个给名字的新方法.

```
Scheme] (letrec ((fac (lambda (x)
                        (if (<= x 0)
                            1
                            (* x (fac (- x 1)))))))
          (fac 10))
   3628800
```

```
Scheme] (letrec ((x y)
                 (y x))
          x)
```

```
  Variable used before given a value: y
```

[eq?]

如果对于计算器的行为还有不确定的地方, 可以参照 R^{n}RS 标准或一个给定的 Scheme 实现.

2 代码

我们代码的大概框架是 `string` → `Syntax` → `Expr` → `Value` → `string`. 没有人规定这里的步骤可以合并或者进一步拆分, 我们只是武断地采取了我们认为比较清晰的做法.

给定需要计算的表达式的字符串, 我们先把它处理成一颗树. 事实上, 本身我们就想写树的, 可惜文字只能是线性的, 我们不得不用括号来表达树的结构. 同时, 我们还简单地判断了一个不含括号的序列是想表达什么. 简单地说, 我们先试图把它理解为整数或者布尔值, 其余的再归一类. 这样的后果是, 我们有很宽泛的命名空间.

```
Scheme] (let ((let (lambda (x y) y))
              (lambda (lambda (x) x)))
         (let lambda (lambda 1)))
```

```
  1
```

然后, 我们把这个树进一步归类. 其中一个重要的步骤是处理上面关键字和变量重名的问题. 到这一步我们已经知道了一个表达式的精确结构.

最后我们对表达式进行求值. 实际上, 用直接代入的做法来求值的话, `Value` 完全可以等于 `Expr`; 但是我们会看到在我们的 (也是广泛采用) 做法中, 为什么要区别这两者.

虽然有争议, 但是我们建议采用面向对象风格的编程方式以锻炼 C++ 技巧. 比如, 我们想求值加法. 但是, 我们不知道其操作数是否是整数; 我们只知道它是一个 `Value`. 不用类型转换, 我们甚至没法取出其中的整数. 一个做法是, 我们在 `ValueBase` 中定义一个虚函数 `int ToInteger()`, 并且默认行为是报错. 而 `Integer` 类修改其行为为返回其所存储的值. 这样, 我们只需要调用操作数的 `ToInteger` 方法, 如果一切正常就可以得到加法的两个数, 否则就会报错. 当然, 你也可以采用 *visitor pattern* 之类的面向对象特供的设计模式.

3 扩展

本次作业有以下可选的扩展. 这次扩展工作量都较大, 但是分数占比低, 仅作有强烈兴趣的同学的自发探索用. 你只需要实现一个即可, 难度与分数无关.

1. 用计算器的可编程特性, 在计算器中写一个能力相同的计算器. 比如,

   ```
   Scheme] (eval (quote (cons 1 2)))
   ```

   ```
     (1 . 2)
   ```

   你可以在计算器中加入新的语法来简化你的程序.

2. 设法将一个 `Value` 转换为一个 `Expr`. 称这个函数为 *readback*, 你需要保证

   $$\forall v \in \texttt{Value}, \mathit{eval}(\mathit{readback}(v)) = v.$$

   这里的等号比较宽松, 比如两个 `Closure` 中的 `x` 并不需要相同, 只要这个重命名是一致的即可, 比如 $\mathit{Clos}(x, x, \varnothing) = \mathit{Clos}(y, y, \varnothing)$.

   请参考 Normalization by Evaluation 或 typed partial evaluation.

3. 加入 `set-car!`, `set-cdr!` 并实现垃圾回收. 建议查询标记-回收算法.

4. 将计算器的求值策略改为懒惰求值. 整体原则是, 任意一个表达式, 除非我们需要它的值来继续下一步操作, 它就不应该被求值. 比如对于 `(car e)`, 我们只需要求值 `e` 到 `(cons e1 e2)` 的程度, 而不能求值 `e1` 和 `e2`.

   这样做有一个显然的坏处: 如果一个参数在函数体中出现了多次, 那么每次都要重新计算. 解决方法也很简单, 对于同样的一个 `Expr`, 我们需要记忆它的求值结果, 如果需要求值时发现已经求过, 直接返回值即可.