解释器的编写

作用域规则

词法作用域

词法作用域(lexical scope)，即静态作用域(static scope)。在不修改绑定时，作用域在词法解析阶段既确定了，不会改变。对于函数而言，寻找绑定时，检查函数的定义环境。

在采用静态作用域的语言里面，名字与对象的约束完全可以在编译时通过检查程序正文确定，不需要考虑运行时的控制流。典型的情况是，对一个特定的名字，其“当前”约束也就是程序里包围着这个给定点的最近的，其中有与该名字匹配的声明的那个块里建立的约束。

词法作用域的一般规则有最内嵌套作用域规则。

动态作用域

动态作用域(dynamic scope)里，变量的寻找过程发生在程序运行时期。对于函数而言，寻找绑定时，检查函数的执行环境。动态作用域里，函数执行遇到一个符号，会由内向外逐层检查函数的调用链，并打印第一次遇到的那个绑定的值。显然，最外层的绑定即是全局状态下的那个值。

采用动态作用域的语言里，名字和对象间的约束依赖于运行时的控制流，特别是依赖于子程序调用的顺序，遵循动态规则：一个给定名字的当前约束也就是在运行区间最近遇到的，而且还没有由于作用域推出而撤销的那个约束。

两者的区别

简单的说，静态作用域中，变量的值是在变量定义时就已决定了，运行时值保持不变（没有重新赋值的情况下），而在动态作用域中，变量的值与定义无关，由运行时决定。

如果采用动态作用域规则，我们无法给任何使用了非局部变量的程序段确定好可预见的引用环境，（比如在过程 a 中的一个局部变量声明，恰好重新定义了函数 b 里使用的全局变量的名字 c，而后过程 a 又调用了函数 b，在一些语言中，做动态语义检查时，可能会出现类型崩溃错误。）

参考网站

https://www.emacswiki.org/emacs/DynamicBindingVsLexicalBinding

https://stackoverflow.com/questions/463463/dynamic-and-lexical-variables-in-common-lisp

作用域的实现

环境

环境，即指解释器保持有关的名字-值对偶（绑定）的轨迹的储存，更准确来说是全局环境。实际上一个计算过程中完全可能设计若干不同的环境。

动态作用域在每次函数求值的时候都会在这唯一的一个环境里查询或更新。

而静态作用域是每次函数求值的时候都创建一个新的环境，包含了函数定义时候的所能访问到的各种绑定。这个新的环境连同那个函数一起，俗称闭包。

为了简单起见，我们采用的静态作用域。

闭包

闭包(closure)，是一种编程语言特性，它指的是代码块和作用域环境的结合，早期由 scheme 语言引入这一特性，随后几乎所有语言都带有这一特性。为了实现 lexical scoping，我们必须把函数做成“闭包”。闭包是一种特殊的数据结构，它由两个元素组成：函数的定义和当前的环境。

闭包里的自由变量会绑定在代码块上，在离开创造它的环境下依旧生效，而这一点使用代码块的人可能无法察觉。自由变量的绑定也会在函数定义的时候就已经确定。

抽象语法树

编译器前端会分析源代码文本，生成一棵抽象语法树 AST（Abstract Syntax Tree）。因此为了写出合理的解释器，我们需要对抽象语法树进行合理的规定。

变量

变量在 MUPL 中的定义是（var s），其中 s 是 Racket string 类型。对变量最基本的操作，是对它的“绑定”（binding）和“取值”（evaluate）。

```
1. (struct var  (string) #:transparent)  ;; a variable, e.g., (var "foo")
```

常数

常数在 MUPL 中定义为一个整形常数。

```
1. (struct int  (num)    #:transparent)  ;; a constant number, e.g., (int 17)
```

基本语法

结构体 add 将 2 个 MUPL 表达式相加求和，ifgreater 判断两个表达式大小然后求值。

```
1. (struct add  (e1 e2)  #:transparent)  ;; add two expressions
2. (struct ifgreater (e1 e2 e3 e4)    #:transparent) ;; if e1 > e2 then e3 else e4
```

函数

跟一般语言不同，我们的函数只接受一个参数。这不是一个严重的限制，因为在我们的语言里，函数可以被作为值传递，也就是所谓“first-class function”。所以你可以用嵌套的函数定义来表示有两个以上参数的函数，即柯里化(currying)。另外，函数分为 3 个部分，函数名，函数参数，函数体。需要注意的是，函数名为#f 时，即为匿名非递归函数，类似于 lambda 表达式。

```
1. (struct fun  (nameopt formal body) #:transparent) ;; a recursive(?) 1-argument functi
   on
```

函数调用

函数调用和闭包的关系密切相关。要想知道函数调用的值，需要知道函数，传入参数值，函数所处的环境。而函数所处的环境又用闭包储存了起来。

```
1. (struct call (funexp actual)        #:transparent) ;; function call
```

局部绑定

用 mlet 表达式对变量进行局部绑定，类似于 let 表达式。

```
1. (struct mlet (var e body) #:transparent) ;; a local binding (let var = e in body)
```

列表

MUPL 中的 apair 对应于 racket 的 cons。在编译器部分，主要用于构建二元组，实现编译环境的储存。

```
1. (struct apair (e1 e2)     #:transparent) ;; make a new pair
```

二元组操作

fst 用于获取二元组前一个元素，snd 用于获取二元组的后一元素。

```
1. (struct fst  (e)    #:transparent) ;; get first part of a pair
2. (struct snd  (e)    #:transparent) ;; get second part of a pair
```

列表操作

aunit 作为标准 MUPL 列表的结尾，类似于 nil，但需要注意的是（aunit）才是真正的列表结尾，而 aunit 只是一个过程而已。比如将 3、4 构建元组，(apair (int 3) (apair (int 4) (aunit))。

```
1. (struct aunit ()    #:transparent) ;; unit value -- good for ending a list
2. (struct isaunit (e) #:transparent) ;; evaluate to 1 if e is unit else 0
```

闭包

闭包用于储存一个函数以及其当前的环境。在对一个函数求值时，由于不知道参数的实际值，我们返回的是一个闭包。而调用函数求值（call）时，则先要通过对函数求值，得到其闭包（主要是为了知道作用域），然后相当于将参数临时绑定在环境中，对函数体求值。

```
1. ;; a closure is not in "source" programs but /is/ a MUPL value; it is what functions
   evaluate to
2. (struct closure (env fun) #:transparent)
```

求值环境

MUPL 语言的求值环境实质上是 racket 的 list，准确来说是二元组。比如 '(("x".2) ("y".3)) 即是将变量 x 绑定为 2，变量 y 绑定为 3。对变量进行绑定或则接触绑定，可以通过 append 或者 cons 函数实现。当然也可以自定义函数来完成。例如下述程序将 x 绑定 3 写入到求值环境中。

```
1.  (append (list (cons “x” 3)) env))
2.  (cons (cons “x” 3) env)
```

作用域的查找，即是通过对二元组的搜索得到的。下面即是在环境 env 查找字符串 str 所代表的表达式的值。

```
1.  (define (envlookup env str)
2.  (cond [(null? env) (error "unbound variable during evaluation" str)]
3.          [(equal? (car (car env)) str) (cdr (car env))]
4.          [#t (envlookup (cdr env) str)]))
```

解释器的实现

在解释器中，就相当于对 MUPL 进行求值，为了使结果一直，我们人为设置求的值的结果仍然是 MUPL 表达式。

var/int/aunit/closure/fun

值的返回整个表达式自身有 3 种情况：整数，空列表。闭包。而对于一个变量则需要到

当前环境种查找相应的值。对于函数求值，需要加上当前环境形成一个闭包，这样子在函数调用的时候才能确保求找出对应的值。

```
1.      [(var? e) (envlookup env (var-string e))]
2.      [(int? e) e]
3.      [(aunit? e) e]
4.      [(closure? e) e]
5.      [(fun? e) (closure env e)]
```

add/ifgreater

这 2 个函数需要注意的地方是，需要检查 e1 和 e2 值是否值为 int 类型。并且在 ifgreater 函数中，e3 和 e4 两则中至多只有一个表达式被求值。

```
1.          [(add? e)
2.           (let ([v1 (eval-under-env (add-e1 e) env)]
3.                 [v2 (eval-under-env (add-e2 e) env)])
4.             (if (and (int? v1)
5.                      (int? v2))
6.                 (int (+ (int-num v1)
7.                         (int-num v2)))
8.                 (error "MUPL addition applied to non-number")))]
9.
10.         [(ifgreater? e)
11.          (let ([v1 (eval-under-env (ifgreater-e1 e) env)]
12.                [v2 (eval-under-env (ifgreater-e2 e) env)])
13.             (if (and (int? v1)
14.                      (int? v2))
15.                 (if (> (int-num v1) (int-num v2))
16.                     (eval-under-env (ifgreater-e3 e) env)
17.                     (eval-under-env (ifgreater-e4 e) env))
18.                 (error "MUPL ifgreater applied to non-number")))]
```

apair

在任何两个 MUPL 列表中求值得到一个 MUPL 表是合法的，因此在这种情况下没有动态类型检查。

```
1.          [(apair? e)
2.           (apair (eval-under-env (apair-e1 e) env)
3.                  (eval-under-env (apair-e2 e) env))]
```

fst/snd

需要注意的是，在 fst 和 snd 函数中，需要检查递归结果是否为 MUPL 表。

```
1.          [(fst? e)
2.           (let ([pr (eval-under-env (fst-e e) env)])
3.             (if (apair? pr)
4.                 (apair-e1 pr)
5.                 (error "MUPL fst applied to non-apair")))]
6.          [(snd? e)
7.           (let ([pr (eval-under-env (snd-e e) env)])
8.             (if (apair? pr)
```

```
9.                  (apair-e2 pr)
10.                 (error "MUPL snd applied to non-pair")))]
```

isaunit

与其他表达式不一样，isaunit 可以接受任何 MUPL 表达式，因此不需要对其进行动态类型检查。

```
1.          [(isaunit? e)
2.           (let ([v (eval-under-env (isaunit-e e) env)])
3.             (if (aunit? v) (int 1) (int 0)))]
```

mlet

将变量进行局部绑定后加入到当前环境中，然后进行求值

```
1.          [(mlet? e)
2.           (let* ([v (eval-under-env (mlet-e e) env)]
3.                  [newenv (cons (cons (mlet-var e) v) env)])
4.             (eval-under-env (mlet-body e) newenv))]
```

call

函数调用时，首先需要判断函数是否形成一个闭包，还需要判断是否有函数名，如果有，则需要将函数名与函数进行绑定加入到环境中进行求值。

```
1.          [(call? e)
2.           (let ([cl  (eval-under-env (call-funexp e) env)]
3.                 [arg (eval-under-env (call-actual e) env)])
4.             (if (closure? cl)
5.                 (let* ([fn (closure-fun cl)]
6.                        [bodyenv (cons (cons (fun-formal fn) arg)
7.                                       (closure-env cl))]
8.                        [bodyenv (if (fun-nameopt fn)
9.                                     (cons (cons (fun-nameopt fn) cl)
10.                                          bodyenv)
11.                                    bodyenv)])
12.                  (eval-under-env (fun-body fn) bodyenv))
13.                (error "MUPL function call with nonfunction")))]
```

自由变量的引入

事实上，一般函数都存在自由变量。在构建一个闭包,它使用一个环境,就像当前环境但只有变量中自由变量的函数闭包的一部分,(一个自由变量是一个变量,出现在一些函数内部没有被绑定的变量)。

为了避免多次计算，我们通过编写一个函数 compute-free-vars 来实现这一点，它接受一个表达式并返回一个包含自由变量的表达式，即将一个类似于 fun 的表达式改写成 fun-challenge 表达式的函数。而在新的函数定义里面有一个字段 freevars 来存储该函数的自由变量集。为了简单起见，我们只考虑函数才携带有自由变量集（其他表达式，可以通过 thunking 实现），而其他非函数表达式，虽然可能有自由变量，依旧直接在环境中寻找值。。

为了实现 compute-free-vars 我们定义了一个新的结构体 res (e fvs)，其中 fvs 储存了表达式 e 的自由变量。同时需要注意的是，若 e 为函数（fun-chalenge）表达式，则 e 也包含了 fvs 自由变量。这样做的好处是，实现函数的递归统一化。对于非函数式时，自由变量可能储存，但没有意义，返回的是其本身。但在对于函数式时，对其递归，一些非函数的表达式的自由变量可能就有需要包含在返回函数的自由变量里面。

```
(struct fun-challenge (nameopt formal body freevars) #:transparent) ;; a recursive(?) 1-argument function

(define (compute-free-vars e)
  (struct res (e fvs)) ; result type of f (could also use a pair)
    (define (f e)
      (cond [(var? e) (res e (set (var-string e)))]
            [(int? e) (res e (set))]
            [(add? e) (let ([r1 (f (add-e1 e))]
                            [r2 (f (add-e2 e))])
                        (res (add (res-e r1) (res-e r2))
                             (set-union (res-fvs r1) (res-fvs r2))))]
            [(ifgreater? e) (let ([r1 (f (ifgreater-e1 e))]
                                  [r2 (f (ifgreater-e2 e))]
                                  [r3 (f (ifgreater-e3 e))]
                                  [r4 (f (ifgreater-e4 e))])
                              (res (ifgreater (res-e r1) (res-e r2) (res-e r3) (res-e r4))
                                   (set-union (res-fvs r1) (res-fvs r2) (res-fvs r3) (res-fvs r4))))]
            [(fun? e) (let* ([r (f (fun-body e))]
                             [fvs (set-remove (res-fvs r) (fun-formal e))]
                             [fvs (if (fun-nameopt e)
                                      (set-remove fvs (fun-nameopt e))
                                      fvs)])
                        (res (fun-challenge (fun-nameopt e) (fun-formal e)
                                            (res-e r) fvs)
                             fvs))]
            [(call? e) (let ([r1 (f (call-funexp e))]
                             [r2 (f (call-actual e))])
                         (res (call (res-e r1) (res-e r2))
                              (set-union (res-fvs r1) (res-fvs r2))))]
            [(mlet? e) (let* ([r1 (f (mlet-e e))]
                              [r2 (f (mlet-body e))])
                         (res (mlet (mlet-var e) (res-e r1) (res-e r2))
                              (set-union (res-fvs r1) (set-remove (res-fvs r2) (mlet-var e)))))]
            [(apair? e) (let ([r1 (f (apair-e1 e))]
                              [r2 (f (apair-e2 e))])
```

```
                                (res (apair (res-e r1) (res-e r2))
                                     (set-union (res-fvs r1) (res-fvs r2)))]
                [(fst? e) (let ([r (f (fst-e e))])
                            (res (fst (res-e r))
                                 (res-fvs r)))]
                [(snd? e) (let ([r (f (snd-e e))])
                            (res (snd (res-e r))
                                 (res-fvs r)))]
                [(aunit? e) (res e (set))]
                [(isaunit? e) (let ([r (f (isaunit-e e))])
                                (res (isaunit (res-e r))
                                     (res-fvs r)))]))
      (res-e (f e)))

(define (eval-under-env-c e env)
  (cond
        [(fun-challenge? e)
         (closure (set-map (fun-challenge-freevars e)
                           (lambda (s) (cons s (envlookup env s))))
                  e)]
         ; call case uses fun-challenge as appropriate
         ; all other cases the same
        ...)

(define (eval-exp-c e)
(eval-under-env-c (compute-free-vars e) null))
```

MUPL 语言的简化

考虑到列表何局部变量都可以用特定的函数表示，用 racket 语言变成如下（函数形式的列表和局部变量）（相当于对柯里化函数（f x y）进行操作）：

```
1. (define (pair x y) (lambda(f) ((f x) y)))
2. (define (fst e)  (e (lambda(x) (lambda(y) x))))
3. (define (snd e)  (e (lambda(x) (lambda(y) y))))
```

但在实际实现 MUPL 语言的条件下，我们需要用 MUPL 语言的函数代替上述语法。

```
1. (define (simplify e)
2.   (cond [(mlet? e)
3.          (call (fun #f (mlet-var e) (simplify (mlet-body e))) (simplify (mlet-e e)))]
4.         [(apair? e)
5.          (fun #f "_f" (call (call (var "_f") (simplify (apair-e1 e))) (simplify (apair-e2
    e))))]
```

```
6.          [(fst? e)
7.           (call (simplify (fst-e e)) (fun #f "_x" (fun #f "_y" (var "_x"))))]
8.          [(snd? e)
9.           (call (simplify (snd-e e)) (fun #f "_x" (fun #f "_y" (var "_y"))))]
10.         [(add? e)
11.          (add (simplify (add-e1 e)) (simplify (add-e2 e)))]
12.         [(ifgreater? e)
13.          (ifgreater (simplify (ifgreater-e1 e))
14.                     (simplify (ifgreater-e2 e))
15.                     (simplify (ifgreater-e3 e))
16.                     (simplify (ifgreater-e4 e)))]
17.         [(fun? e)
18.          (fun (fun-nameopt e) (fun-formal e)
19.               (simplify (fun-body e)))]
20.         [(call? e)
21.          (call (simplify (call-funexp e))
22.                (simplify (call-actual e)))]
23.         [(isaunit? e)
24.          (isaunit (simplify (isaunit-e e)))]
25.         [#t e]))
```

推荐阅读：

一、函数面面观

二、括号神教

三、语言和同像性

四、编译器与解释器

五、动态作用域

六、词法作用域和闭包

七、first-class continuation

八、尾调用与 CPS

九、For Great Good

十、类型和类型系统

十一、Pure and Lazy

十二、多态性

十三、Weak head normal form