CS61A NOTE12 Scheme

Primitives and Defining Variables 初始和定义变量

Scheme has a set of **atomic** primitive expressions. Atomic means that these expressions cannot be divided up.Scheme 有原子性的原始表达式。原子:这些表达式不能被分割。

纯文本

```
scm> 123
123
scm> #t
True
scm> #f
False
```

Unlike in Python, the only primitive in Scheme that is a false value is `#f` and its equivalents, `false` and `False`. **This means that 0 is not false.**

与之前不同，Scheme 中唯一假值是 #f /false /False

In Scheme, we can use the `define` special form to bind values to symbols, which we can then use as variables. When a symbol is defined this way, the `define` special form returns the symbol.在 Scheme 中可以用 define 特殊形式将值绑定到 symbols 上，然后我们可以将其作为变量使用。当 symbols 以这种方式被定义时，define 返回该 symbol。

- `(define <variable name> <value>)`

Evaluates `<value>` and binds the value to `<variable name>` in the current environment.算值并绑定到变量名(当前环境都用这个)直接输出是输出变量名

WWSD

纯文本

```
scm> (define a 1)
a
scm> a
1
```

```
scm> (define b a)
b
scm> b
1 #不是a，a已经是1
scm> (define c 'a)
c
scm> c
a       #'a 不是1，是字符
```

Call Expressions

Call expressions apply a procedure to some arguments.

纯文本

```
(<operator> <operand1> <operand2> ...)
```

Call expressions in Scheme work exactly like they do in Python. To evaluate them:

1. Evaluate the operator to get a procedure.用 operator 操作符
2. Evaluate each of the operands from left to right.从左到右用 operands
3. Apply the value of the operator to the evaluated operands.应用 operator 在 operands 上

For example, consider the call expression `(+ 1 2)`. First, we evaluate the symbol 符号 `+` to get the built-in addition procedure 内置加法. Then we evaluate the two operands `1` and `2` to get their corresponding atomic values 原子值. Finally, we apply the addition procedure to the values `1` and `2` to get the return value `3`.应用操作符在 12 上

Operators may be symbols, such as `+` and `*`, or more complex expressions, as long as they evaluate to procedure values.操作可以是符号，如 + 和*，也可以是更复杂的表达式，只要它们评估为过程值。

纯文本

```
scm> (- 1 1)                ; 1 - 1
0
scm> (* (+ 1 2) (+ 1 2))    ; (1 + 2) * (1 + 2)
9
```

WWSD

What would Scheme display? As a reminder, the built-in `quotient` function performs floor division.内置的商函数可以执行地板除法⌊⌋//。

纯文本

```
scm> (define a (+ 1 2)) #define直接输出为本身
a
scm> a
3
scm> (define b (- (+ (* 3 3) 2) 1))
b
scm> (+ a b)
13
scm> (= (modulo b a) (quotient 5 3)) #前者mod 1，后者//，1
#t
```

Special Forms

Special form expressions contain a **special form** as the operator. Special form expressions *do not* follow the same rules of evaluation as call expressions. Each special form has its own rules of evaluation -- that's what makes them special! Here's the Scheme Specification to reference the special forms we will cover in this class.特殊形式表达式包含一个特殊形式作为运算符。特殊形式表达式不遵循与调用表达式相同的评估规则。每个特殊形式都有自己的求值规则

It is important to note that everything in Scheme is either an **atomic** or an **expression**, so although these special forms look and operate similarly to Python, they are evaluated differently.需要注意的是，Scheme 中的所有东西要么是**原子**，要么是**表达式**，所以尽管这些特殊形式的外观和操作与 Python 相似，但它们的求值方式却不同。

Special forms like `if`, `cond`, `and`, `or` in Python direct the control flow of a program and allow you to evaluate specific expressions under some condition. In Scheme, however, these special forms are expressions that take in a set amount of parameters and return some value based on the condition passed in.像 Python 中的 `if`, `cond`, `and`, `or` 这样的特殊形式可以引导程序的控制流，并允许你在某些条件下评估特定的表达式。但在 Scheme 中，这些特殊形式是表达式，它接收了一定数量的参数，并根据传入的条件返回一些值。

If Expression(选前面/后面)

An `if` expression looks like this:

`(if <predicate> <if-true> [if-false])`

`<predicate>` and `<if-true>` are required expressions and `[if-false]` is optional.前两者必须，后一个可选

The rules for evaluation are as follows:

1. Evaluate `<predicate>`.
2. If `<predicate>` evaluates to a truth-y value, evaluate `<if-true>` and return its value. Otherwise, evaluate `[if-false]` if provided and return its value.如果<predicate>求值为真，则求<if-true>并返回。否则，如果提供[if-false]，就求值并返回。

`if` is a special form as not all of its operands will be evaluated. The value of the first operand determines whether the second or the third operator is evaluated.if 是一个特殊的形式，因为不是所有的操作数都会被评估。第一个操作数的值决定了第二个或第三个操作数是否被评估。Only #f is a false-y value in Scheme; everything else is truth-y, including 0.在 Scheme 中，只有#f 是假 Y 值，其他都是真 Y 值，包括 0

纯文本

```
scm> (if (< 4 5) 1 2)
1
scm> (if #f (/ 1 0) 42)
42
```

Cond Expression 条件表达式(直到找到真值)

A `cond` expression looks like this:

`(cond (<pred1> <if-pred1>) (<pred2> <if-pred2>) ... (<predn> <if-predn>) [(else <else-expression>)])`

Must have at least one `<predn>` and `<if-predn>` and `[(else <else-expression>)]` is optional.前两项必要后一项不必要

The rules for evaluation are as follows:

1. Evaluate the predicates `<pred1>`, `<pred2>`, ..., `<predn>` in order until you reach one that evaluates to a truth-y value.直到找到真值

2. If you reach a c that evaluates to a truth–y value, evaluate and return the corresponding expression in the clause.

3. If none of the predicates are truth–y and there is an else clause, evaluate and return `<else-expression>`.如果没真，返回<else–expression>

`cond` is a special form because it does not evaluate its operands in their entirety; the predicates are evaluated separately from their corresponding return expression. In addition, the expression short circuits 短路 upon reaching the first predicate that evaluates to a truth–y value, leaving the remaining predicates unevaluated.第一个真值短路，剩下不 value

纯文本

```
scm> (cond #类似if elif elif......
        ((< 4 5) 1)
        (else 2)
      )
1
scm> (cond
        (#f (/ 1 0))
        (else 42)
      )
42
```

Let Expressions(返回最后值)

A `let` expression looks like this: `(let ([binding_1] ... [binding_n]) <body> ...)`

Each `binding` corresponds to expressions of the form `(<name> <expression>)`.

Scheme evaluates a `let` expression using the following steps:

1. Create a new local frame that extends the current environment (in other words, it creates a new child frame whose parent is the current frame).创建一个新的本地框架/它创建了创建一个新的本地框架一个新的子框架，其父级是当前框架)

2. For each `binding` provided, bind each `name` to its corresponding evaluated `expression`.绑定
3. Finally, the `body` expressions are evaulated in order in this new frame, returning the result of evaluating the last expression.按顺序算，返回最后值

Note that bindings are optional within a `let` statement, but we typically include them

纯文本

```
scm> (let (
            (x 5)
            (y 10)
          )
      (print x)
      (print y)
      (- x y)
      (+ x y)
     )
5
10
15 #返回最后值
```

Note that `(- x y)` in the body of this `let` expression *does* get evaluated, but the result doesn't get returned by the `let` expression because only the value of the *last* expression in the body, `(+ x y)`, gets returned. Thus, the interpreter does *not* display `-5` (the result of `(- x y)`).(– x y)被计算，但并没有被 let 表达式返回，只有主体中最后一个表达式的值(+ x y)被返回。

However, we see that `5` and `10` *are* displayed out by the interpreter. This is because printing `5` and printing `10` were *side effects* of evluating the expressions `(print x)` and `(print y)`, respectively. `5` and `10` are *not* the return values of `(print x)` and `(print y)`.打印 5 和打印 10 是打印 x 和打印 y 的副作用，5 和 10 不是 print x 和 print y 的返回值

Begin Expressions(返回最后值)

A `begin` expression looks like this: `(begin <body_1> ... <body_n>)`

Scheme evaluates a `begin` expression by evaluating each `body` in order in the current environment, returning the result of evaluating the last `body` 方案通过在当前环境中依次评估每个体来 value，返回最后一个 body 的结果

纯文本

```
scm> (begin
        (print (< 2 3))
        (print 'hello)
        (+ 1 2)
        (- 5 7)
     )
#t
hello
-2
```

(+1 2)确实被计算了，但是结果 3 并没有被 begin 表达式返回（因此也没有被解释器显示），因为它不是最后一个主体表达式

Boolean operators

Like Python, Scheme has the boolean operators `and`, `or`, and `not`. `and` and `or` are special forms because they are short-circuiting operators, while `not` is a builtin procedure.短路运算符：and or；内置过程：not

- `and` takes in any amount of operands and evaluates these operands from left to right until one evaluates to a false-y value. It returns that first false-y value or the value of the last expression if there are no false-y values.返回第一个假 Y 值，如果没有假 Y 值返回最后一个表达式的值。
- `or` also evaluates any number of operands from left to right until one evaluates to a truth-y value. It returns that first truth-y value or the value of the last expression if there are no truth-y values.返回第一个真-Y 值，如果没有真-Y 值返回最后一个表达式的值
- `not` takes in a single operand, evaluates it, and returns its opposite truthiness value.接收一个操作数，评估，并返回相反值

纯文本

```
scm> (and 25 32)
32
scm> (or 1 (/ 1 0))    ; Short-circuits
1
scm> (not (odd? 10))
#t
```

WWSD

纯文本

```
scm> (if (or #t (/ 1 0)) 1 (/ 1 0)) #看清括号，最后两个是前/后，or后是两个只执行第
1

scm> ((if (< 4 3) + -) 4 100)
-96
```

纯文本

```
scm> (cond #到真
        ((and (- 4 4) (not #t)) 1)
        ((and (or (< 9 (/ 100 10)) (/ 1 0)) #t) -1) #or执行前者，and全对，输出
        (else (/ 1 0))
    )
-1
```

纯文本

```
scm> (let (
            (a (- 3 2))
            (b (+ 5 7))
          )
        (* a b)     不输出
        (if (< (+ a b) b)
            (/ a b)
            (/ b a)   只输出最后
        )
    )
>>>12
```

纯文本

```
scm> (begin
        (if (even? (+ 2 4)) #对
            (print (and 2 0 3)) #执行此行，3
            (/ 1 0)
        )
        (+ 2 2) #begin不return
        (print 'lisp) #lisp，side effect
        (or 2 0 3) #begin只return此行2
```

```
    )

3
lisp

2
```

Defining Functions

All Scheme procedures are constructed as lambda procedures.

One way to create a procedure is to use the `lambda` special form.

`(lambda (<param1> <param2> ...) <body>)`

This expression creates a lambda function with the given parameters and body, but does not evaluate the body. As in Python, the body is not evaluated until the function is called and applied to some argument values. The fact that neither the parameters nor the body is evaluated is what makes `lambda` a special form.

We can also assign the value of an expression to a name with a `define` special form:

1. `(define (<name> <param> ...) <body> ...)`
2. `(define <name> (lambda (<param> ...) <body> ...))`

These two expressions are equivalent; the first is a concise version of the second.

纯文本

```
scm> ; Bind lambda function to square
scm> (define square (lambda (x) (* x x)))
square

scm> (define (square x) (* x x))          ; 直接lambda写成square
square

scm> square
(lambda(x) (*x x)) #注意最外括号

scm> (square 4)
16
```

lab11 WWSD

纯文本

```
scm> (/ 8 2 2)
2

scm> (quotient 29 5)
5

scm> (= 1 3)      ; Scheme uses '=' instead of '==' for comparison
#f

scm> (x y) #x是14，y是7
14      #x是lambda函数

scm> (if (not (print 1)) (print 2) (print 3))
1
3 #不print2，跳过了

scm> (define foo (lambda (x y z) (if x y z)))
scm> (foo 1 2 (print 'hi))
hi  #先evaluate再传入
2

scm> ((lambda (a) (print 'a)) 100) #100输入，但'a还是输出a
a
```

disc Q1: Virahanka-Fibonacci

Write a function that returns the `n` -th Virahanka-Fibonacci number.

纯文本

```
(define (vir-fib n)
    'YOUR-CODE-HERE
    (if (<= n 1)
        n #不用加括号时别加，且不是1，n=1时1，n=0时0，不用定义n=2
        (+ (vir-fib (- n 1)) (vir-fib (- n 2)))
    )
)
```

```
(expect (vir-fib 10) 55)
(expect (vir-fib 1) 1)
```

Q2: `fib2` (Su21 Final #5a) Solution

Each element of the **fibonacci2** sequence is defined as twice the absolute value of the difference between the previous two elements. Assume that the 0th element of the **fibonacci2** sequence is 0, and the 1st element is 1. Implement the function **`fib2`**, which takes in one parameter **n**, a non-negative integer, and returns the nth element of the **fibonacci2** sequence. Reminder: Scheme has a built in procedure **abs** which returns the absolute value of the argument that is passed in.

```
(define (fib2 n)
    (if (<= n 1) n
        (* 2 (abs (- (fib2 (- n 1)) (fib2 (- n 2)))))
    )
)
```

lab11 Q2 over or under

Define a procedure `over-or-under` which takes in a number `num1` and a number `num2` and returns the following:

- -1 if `num1` is less than `num2`
- 0 if `num1` is equal to `num2`
- 1 if `num1` is greater than `num2`

纯文本

```
(define (over-or-under num1 num2)
     (cond
         ((< num1 num2) -1)
         ((= num1 num2) 0)
         ((> num1 num2) 1)
     )
)
```

lab11 Q3: Make Adder

Write the procedure `make-adder` which takes in an initial number, `num`, and then returns a procedure. This returned procedure takes in a number `inc` and returns the result of `num + inc`.提示用 lambda 函数 inc 是输入的

> *Hint*: To return a procedure, you can either return a `lambda` expression or `define` another nested procedure. Remember that Scheme will automatically return the last clause in your procedure.
>
> You can find documentation on the syntax of `lambda` expressions in the 61A scheme specification!

纯文本

```
(define (make-adder num)
    (lambda(x)(+ x num))
)
```

lab11 Q4: Compose

Write the procedure `composed`, which takes in procedures `f` and `g` and outputs a new procedure. This new procedure takes in a number `x` and outputs the result of calling `f` on `g` of `x`.返回 f(g(x))没给 x 提示用 lambda

纯文本

```
(define (composed f g)
    (lambda(x) (f(g x))
    )
)
```

lab11 Q5: Repeat

Write the procedure `repeat`, which takes in a procedure `f` and a number `n`, and outputs a new procedure. This new procedure takes in a number `x` and outputs the result of applying `f` to `x` a total of `n` times. For example:

scm> (define (square x) (* x x))

square

scm> ((repeat square 2) 5) ; (square (square 5))

625

scm> ((repeat square 3) 3) ; (square (square (square 3)))

6561

scm> ((repeat square 1) 7) ; (square 7)

49

> *Hint:* The `composed` function you wrote in the previous problem might be useful.

纯文本

```
(define (repeat f n)
 (if (< n 1)
    (lambda(x) x)
    (composed f (repeat f (- n 1) ))
 )
)
```

lab11 Q6: Greatest Common Divisor

The GCD is the the greatest common divisor of two numbers.

Write the procedure `gcd`, which computes the GCD of numbers `a` and `b`. Recall that Euclid's Algorithm tells us that the GCD of two values is either of the following:

- the smaller value if it evenly divides the larger value, or
- the greatest common divisor of the smaller value and the remainder of the larger value divided by the smaller value

In other words, if `a` is greater than `b` and `a` is not divisible by `b`, then

gcd(a, b) = gcd(b, a % b)

纯文本

```
(define (max a b)
  (if (> a b)
```

```
        a
        b))

(define (min a b)
  (if (> a b)
      b
      a))

(define (gcd a b)
  (cond
     ((zero? a) b)
     ((zero? b) a)
     ((= (modulo (max a b) (min a b)) 0) (min a b))
     (else (gcd (min a b) (modulo (max a b) (min a b) ) ))
  )
) #别多写括号
```

lab12 Q6: Count

Implement `count`, which takes in an element `x` and a list `s` and returns the number of times that `x` is contained in `s`.

纯文本

```
scm> (count 1 '(1 2 3 4))
1
scm> (count 'a '(a b a b a))
3

(define (count x s)
  (if (null? s) 0  #递归定义null
      (+ (if (eqv? x (car s)) 1 0) #很聪明的做法，类似1函数
         (count x (cdr s))))
)
```

Scheme Lists - Useful Functions

`(length <list>)`

- Returns the number of elements in **`list`**.
- **`(length (list 1 2 3 4 5)) -> 5`**
- **`(length (list 1 2 (list 3 4) 5)) -> 4`**

`(append <list1> <list2>)`

- Returns a new list containing the elements of **`list1`**, then the elements of **`list2`** in that order.
- **`(append (list 1 2 3 4) (list 5 6 7 8)) -> (1 2 3 4 5 6 7 8)`**

Scheme Lists - Useful Functions

`(map <func> <list>)`

- Returns a new list containing the elements of **`list`** in the same order but with **`func`** applied to them
- **`(map even? (list 1 2 3 4)) -> (#f #t #f #t)`**

`(filter <func> <list>)`

- Returns a new list that only contains the elements of **`list`** that return true when **`func`** is called on them
- **`(filter odd? (list 1 2 3 4)) -> (1 3)`**

lambda 提醒用 map+append

Q4: `join` (Fa20 Final #5b)

The **`join`** procedure takes two lists of lists **s** and **t**. It returns a list of lists that has one element for each possible pairing of an element of **s** with an element of **t**. Each element of the result is a list that has all the elements of a list from **s** followed by all the elements of a list from **t**.

```
scm> (define instructors '(
        (john 61a)
        (hany 61a)
        (josh 61b)))
instructors
scm> (define grades '(
        (a b)
        (c d)))
grades
scm> (join instructors grades)
((john 61a a b) (john 61a c d) (hany 61a a b) (hany 61a c d) (josh 61b a b) (josh 61b c d))
```

Implement join.

```
(define (join s t)

  (if (null? s) nil

      (_________ (_________ (lambda (v) (_________ _________ _________)) t)
      ;   (a)         (b)                    (c)       (d)       (e)

            (join _________ t))))
            ;        (f)
```

Q4: `join` (Fa20 Final #5b) Solution

The **`join`** procedure takes two lists of lists **s** and **t**. It returns a list of lists that has one element for each possible pairing of an element of **s** with an element of **t**. Each element of the result is a list that has all the elements of a list from **s** followed by all the elements of a list from **t**.

```scheme
(define (join s t)
    (if (null? s) nil
        (append (map (lambda (v) (append (car s) v)) t)
        ; (a)   (b)                      (c)     (d) (e)
        (join (cdr s) t))))
              ; (f)
```

cons 递归

Q5: countdown (Su21 #5b) Solution

```scheme
(define (countdowns lst)
    (cond ((null? lst) nil)
                      ;(k)
          ((> (car lst) 0) (cons (car lst)
                  ;(l))
            (countdowns (cons (- (car lst) 1) (cdr lst)))))
                                ; (m)
          (else (cons 0 (countdowns (cdr lst))))))
                       ; (o)

(expect (countdowns '(3)) (3 2 1 0))
(expect (countdowns '(2 0 3)) (2 1 0 0 3 2 1 0))
(expect (countdowns '()) ())
```

lab12 Q7: Unique

Implement `unique`, which takes in a list `s` and returns a new list containing the same elements as `s` with duplicates removed.

纯文本

```scheme
scm> (unique '(1 2 1 3 2 3 1))
(1 2 3)
scm> (unique '(a b c a a b b c))
(a b c)

(define (unique s)
  (if (null? s) nil
      (cons (car s)
```

```
            (unique (filter (lambda (x) (not (eqv? (car s) x))) (cdr s)))))
)
```

lab12 Q8: Tally

Implement `tally`, which takes a list of `names` and returns a list of 2-element lists, one for each unique name in `names`. Each 2-element list should contain a name and the number of times that the name appeared in `names`. Each name should appear only once in the output, and the names should be ordered by when they first appear in `names`.

Hint: You might want to use `count` and `unique` procedures in your solution. You may also want to use `map`.

```
纯文本
scm> (tally '())
()
scm> (tally '(kevin irene ashley irene irene ashley brandon irene irene bra
((kevin 1) (irene 5) (ashley 2) (brandon 2))


(define (tally names)
  (map (lambda (name) (list name (count name names))) (unique names))
)
```