Racket - P10

general infix notation

注意通常来说 dot notation 仅允许当有 .(…) 的形式，或 dot 在序列的最后一个元素之前
但是也允许一对 dot 存在于一个单独的元素两侧的情况，reader 会将这个元素放在序列开头
如 (1. < . 2) → (< 1 2) → #t，'(1. < . 2) → '(< 1 2)（注意这个元素不能是第一个或最后一个）
但也要注意这种表示是 non-traditional 的，对于 non-list pair 也没意义

macro - rewriting system for syntax
(if cond then trueClause else falseClause) → (if cond trueClause falseClause)
注意 if 并不是 function call，因为在判断前不需要 evaluate True/False Clause
(define-syntax my-if
(syntax-rules (then else)
[(my-if e1 then e2 else e3)
(if e1 e2 e3)]))

其语法结构为 (define-syntax <id>
(syntax-rules (<id>*)
{[(syntax pattern) (Racket pattern)]}+))

Boolean

在 Racket 中，Bool 类型值为 #t (true) 和 #f (false)，大写 #T/#F 也可用.
注意 Racket 中所有 non-false value 都按 #t 处理

Numbers

在 Racket 中，number 分为 exact number 和 inexact number
exact number：包括任意大小的整数 (integer)，如 99999999999999 或 -11，
以分数形式 (整数分子/分母) 表示的有理数 (a rational exactly ratio of two integers)
实部和虚部都是精确的复数 (complex with exact real and imaginary parts)
inexact number：IEEE 浮点数或是无穷大/非数字，(IEEE floating-point infinities, not-a-number)
实部或虚部是非精确的复数 (complex with IEEE floating-point real and/or imaginary parts)
exact 以整数和分数形式 print，inexact 以 decimal point 或 exponent specifier 的形式 print
如 exact：-3/4，1+2/3i，以及 inexact：3.14e+87，-inf.0，+nan.0，1.0+2i，-inf.0-nan.0i

#e / #i

用于声明输入的立即数被视为 exact (#e) 或 inexact (#i)
即如 #e11.23+7.11i → 1123/100+711/100i，#i 0+1/10i → 0.0+0.1i

#b / #o / #x

用于声明按二进制 (#b)，八进制 (#o)，十六进制 (#x) 来解释 (interprete) 输入的立即数
如 #b1011 → 11，#o1011 → 521，#x1011 → 4113

注意：只要计算中出现 inexact number，则计算结果为 inexact，所以可视为一种"污染"
Racket 中提供 inexact->exact 和 exact->inexact 函数用于在 exact 和 inexact 之间转换
特别注意 (inexact->exact) 和 #e 采用不同的方法，可能产生不同结果
如 #e0.5 → 1/2，但 (inexact->exact 0.5) → 1/2
#e0.1 → 1/10 (inexact->exact 0.1) → 3602879701896397/36028797018963968

Racket - P11

在Racket中，只能表示有理数或是实部虚部都是有理数的复数

函数 sqrt, log, sin 等会产生无理数结果，如 (sqrt 2) → 1.4142135623730951

在Racket中，提供部分 procedure 来判断是否某一类型的数值，

如：number?, integer?, rational?, real?, complex

数值　整数　有理数　实数　复数

= 和 eqv?

注意在Racket中，= 和 eqv?（以及 equal?）的处理方式是不同的。

对于 =，如果比较 exact 和 inexact，则会先将 inexact 转换为 exact 再进行比较

注意在将 inexact 转换为 exact 的过程中，可能出现预期外的结果

如 (= 1/2 0.5) → #t，(= 1/10 0.1) → #f，(inexact -> exact 0.1) → $\frac{3602879701\cdots}{36028797014\cdots}$

对于 eqv? 和 equal?，会判断 exactness 和 numerical equality

如 (equal? 1 1.0) → #f，(equal? 1/2 0.5) → #f

character

在Racket中，character 对应于 Unicode 标量值（scalar value），

其中 scalar value 是一个 21位无符号整型值（21-bit unsigned integer）

scalar value 对应到自然语言中的字符或一串字符（a natural-langage character or piece）

scalar value 在Unicode标准中，是比 "character" 概念更简单的记法

如罗马数字可以表示成一个 scalar value

char->integer 和 integer->char 用于整型值和字符间相互转换

如 (char->integer #\I) → 73，(integer->char 121) → #\y

可打印（printable）会显示为 #\ 加上 represented character，如 #\I，#\入

不可打印（unprintable）显示为 #\u 加上 十六进制 scalar value，如 #\u1FFF

特殊字符 显示为 #\ 加上特殊字，如 #\space，#\newline

display

用于将 character 打印在 current output port 中，如 (display #\u03BE) → ξ

Racket 中提供 procedure 来判断字符的分类

如 char-alphabetic?（字母），char-numeric?（数字），char-whitespace?（空白符）

char->upcase 和 char->downcase 用于大小写转换，如 (char->downcase #\A) → #\a

注意 大小写转换并不适用于非英文的自然语言字符，如 (char->upcase #\β) → #\β

Racket 中提供 procedure 来判断字符是否相同 → #f

char=? 等价于 eqv? 和 equal? 作用于 character，比较的是 scalar value，如 (char=? #\A #\a)

char-ci=? 则会忽略字符的大小写，如 (char-ci=? #\A #\a) → #t

Racket - P12

Lexing

Source Code ⟶ Lexer / Tokenizer (Regular Languages Expressions) ⟶ Stream of Tokens (Words)

- recognizing words in a language (Programming / Formal / Natural)

Lexer - Library / Functions

eg (+ 2 3) eof =>

token-LeftParen token-plus (token-Numeric 2) (token-Numeric 3) token-RightParen

"empty" "value"

eg Language — Numeric and Boolean Expression

```
(require parser-tools/lex
         (prefix-in : parser-tools/lex-sre))
(provide (all-defined-out))
```

类似于C++的head file，声明引用用的头文件或者模块 (Python和Haskell的import)

provide允许模块在被require的地方可获取 all-defined-out 简写为输出所有在输出模块中定义的绑定

```
(define-empty-tokens parens (LEFTPAREN RIGHTPAREN))
(define-empty-tokens bool-operators (AND OR NOT))
(define-empty-tokens math-operators (PLUS MULTIPLY))
(define-empty-tokens comparison-operators (SAME NOTSAME SMALLER NOTSMALLER))
(define-empty-tokens if-keywords (IF THEN ELSE))
(define-empty-tokens end-of-file (EOF))
```

定义"empty"的token，即此类token仅具有一个token name类型，而没有value

```
(define-tokens names-and-values (NUMERIC BOOLEAN))
```

定义"value"类的token，注意不用写value而直接define-tokens

此类token在类型外还包括一个value，生成token时必须定义一个value

```
(define mylexer (lexer
  [whitespace (mylexer input-port)]
  [#\( (token-LEFTPAREN)]
  [(:or "AND" "And") (token-AND)]
  [(:+ numeric) (token-NUMERIC (string->number lexeme))]
  [(eof) (token-EOF)]))
```

这部分实际定义了从str到token的生成方式

注意这部分使用的是pattern matching，在匹配到一个clause后，使用clause最后一个<expr>

:or表示"或"，即满足其一，:+表示"至少一个"，类似于a+

#\C是character表示(eof)表示到达结尾，whitespace表示空白符

```
(define (get-tokenizer in) (λ () (mylexer in)))
```

定义get-tokenizer为一个lambda函数

注意Racket中可以用入指代lambda

```
(define (lex in) (let ([tokenizer (get-tokenizer in)])
                   (define (lex-function) (let ([tok (tokenizer)])
                     (cond [(eq? tok (token-EOF)) null]
                           [else (cons tok (lex-function))])))
                   (lex-function)))
(define (lexstr str) (lex (open-input-string str)))
```

tok为下一个生成的token

如果token为EOF，则为(空list)null

如果不是，则返回一个cons包含当前token和递归调用

token的列表并返回

对于输入的字符串string，分别为

Racket - P13

Parsing

Stream of Tokens → Parser → "sentences" (Expressions) → Intermediate / Representation
(Context Free Languages) (Abstract Syntax Tree)

Context Free Grammar : Backus-Naur Form (BNF) start symbol : <program>

<program> ::= <booleanExpr> | <numericExpr> | <ifExpr>
<booleanExpr> ::= [redacted] BOOLEAN | <compExpr> | (AND / OR / NOT <booleanExpr> <booleanExpr>)
<compExpr> ::= (SAME / NOTSAME <program> <program>) | (SMALLER / NOTSMALLER <program> <program>)
<ifExpr> ::= (IF <booleanExpr> THEN <program> ELSE <program>)
<numericExpr> ::= [redacted] NUMERIC | (PLUS / MULTIPLY <numericExpr> <numericExpr>)

自顶向下(Top Down) : LL(1) , 自底向上(Button Up) : LR(1)

Grammar (Context Free)
- Rules : Nonterminal Symbols ⇒ Sequences of Nonterminal / Terminal (产生式的集合)

Env. 环境 Environment
- Track variables and their associated values
- Data Structure
- Procedural / Functionalized implementation

以DBV实现中的environment为例

```
(define (apply-env env [redacted] var)
  (let ([res (assoc var env)])
    (if res (cdr res) #f)))
```

注意：env应存储结构为(sym, memref)pairs的list

empty-env : create an empty env.
extend-env : add a new binding
apply-env : lookup a variable's value

```
(define (extend-env env k v)
  (let ([ref (memref k v)])
    (cons (cons k ref) env)))
```

将k,v组装为memref结构

将(sym . memref)加入env 即一个key-value

(struct memref (sym [value] #:mutable) #:transparent) [redacted]定义了变量的基本结构. 对

注意这里 #:transparent 表示此结构对于其他 module 是透明的(内部可见)的.
struct 默认是 opaque(不透明的), 即只有这个 module 中的 accessors 和 mutator 可访问内部

注意 #:mutable 表示 memref-value 是可变的, 即可以使用 set-memref-value! 命令

即有 (define (deref ref)
  (memref-value ref))
用于从 memref 结构中提取 value

(define (setref! ref val)
  (set-memref-value! ref value))
用于将[redacted]传入的 memref 结构中的 value 置为 val

(define ([redacted]empty-env) '()) 定义一个 empty-env 函数, 返回一个空环境(即一个空list)

Racket - P14

function apply (function name parameter)

- find definition of function in the environment
- evaluate parameter in the current environment
  - add a new binding formal paramater to the actual parameter value
- evaluate function body in that new environment

```
(define myparser
  (parser
    (start prog)          定义起始符
    (end EOF)             定义token流的结尾
    (tokens name-and-values
            end-of-file
            key-words
            lambda-keywords
            parens ...)   定义出现的token 模式
    (error (lambda (tok-ok? tok-name tok-value)
           (printf "Parser error: token ~a value ~a"
                  tok-name tok-value)
    (grammar
     (prog
      [(LEFT LAMBDA LEFT IDENTIFIER RIGHT prog RIGHT)
       (lambda-expr (identifier-expr $4) $6)]          定义lambda表达式
      [(LEFT LET LEFT IDENTIFIER    如 (let ([f (anonymous function)]) expr)
        LEFT LAMBDA LEFT IDENTIFIER RIGHT prog RIGHT
        RIGHT prog RIGHT)
       (let-expr (identifier-expr $4) (lambda-expr $8 $10) $13)]   定义在let语句中使用匿名函数的结构
      [(LEFT IDENTIFIER prog RIGHT) (function-app $2 $3)]   定义函数调用
      [(NUMERIC) (numeric-expr $1)]                         定义数字
      [(IDENTIFIER) (identifier-expr $1)]))))               定义变量名
```

```
(define (evalHelper expr env)
  (match expr
    定义加法运算
    [(plus e1 e2) (+ (evalHelper e1 env)
                     (evalHelper e2 env))]
    定义let语句运算
    [(let-expr name val e)
     (evalHelper e (extend-env
                     (identifier-expr-arg1 name)     获取变量名
                     (evalHelper val env)            并在当前环境计算val
                     env))]                          结果绑定到运算环境
    定义lamba表达式: 将函数名, 参数, 定义环境封装
    [(lambda-expr para body)
     (closure para body env)]
    定义函数调用
    [(function-app name parameter)
     (let ([funcdef (apply-env env name)])
       (evalHelper (closure-body funcdef)
                   (extend-env (closure-para funcdef)
                               (evalH para env)
                               (closure-env funcdef))))]
    定义立即数
    [(numeric-expr e) e]
    定义从环境获取变量值
    [(identifier-expr e)
     (apply-env env e)]
  ))
```

ast文件中的struct定义 用于eval中的 pattern matching

```
(struct lambda-expr (parameter body))
(struct let-expr (arg1 arg2 arg3))
(struct function-app (name parameter))
(struct closure (parameter body env))
(struct numeric-expr (arg1))
(struct identifier-expr (arg1))
```

```
(define (parse in)
  (myparser (get-tokenizer in)))
(define (parsestr str)
  (let ([in (open-input-string str)]) (parse in)))
```

测试parser 使用字符串给定

```
(define (eval expression)
  (evalHelper expression (empty-env)))
```

用Parser的输出测试evaluate的结果

Racket - P15

match Racket中的match函数用于进行pattern matching，其形式与Haskell中的case相似

syntax为 (match val-expr clause ...)

clause = [pat body ...+] | [pat (=> id) body ...+] | [pat #:when cond-expr body ...+]

注意 clause ... 等同于 clause*，body ...+ 等同于 body+

运算方式与Haskell中的pattern matching类似，根据val-expr的值进行匹配

当遇到第一个匹配时即计算body中的表达式，并将最后一个表达式的值作为match的值

* the last body is evaluated in tail position with respect to the match expression

```
(define (my-map f lst)
  (match lst
    ['() '()]
    [(list x l ...)
     (cons (f x) (my-map f l))]))
```

```
myMap :: (a -> b) -> [a] -> [b]
myMap _ [] = []
myMap f (x:xs) = (f x) : (myMap f xs)

myMap' :: (a -> b) -> [a] -> [b]
myMap' f lst = case lst of
  [] -> []
  (x:xs) -> (f x) : (myMap' f xs)
```

注意这两个程序是等价的，其中↑为Racket ↓为Haskell

另外 ↑ 为用的是对表达式的值进行match

↓ 为用的是对参数又传入进行match

pat ::=

literal 即比较如boolean, number, char, string等数值，等价于equal? constant

_ 表示match anything，与Haskell中相同，相当于占位符并无视该位置的pattern

id / (var id) 也表示match anything，并且将匹配到的值绑定到id，bind id to the matching value

注意 如果在一次match使用了同一个id多次，则所有使用id的位置必须一致

匹配标准为 match-equality-test

如 (match '(1 2 1) [(list a b a) (list a b)] [_ 'else]) → '(1 2)

(cons pat pat) 用于匹配 pair of patterns，且每一项又分别去地看作一个pat

如 (match '(1 (2 3)) [(list _ (and a (list _ ...)) a]) → '(2 3)

(and pat ...) match when all patterns match

(or pat ...) match when any pattern match 如 (match '(1 2) [(or (list a 1) (list a 2)) a]) → 1

(not pat ...) match whe no pattern match 如 (match '(1 4 3) [(list (not 4) ...) 'yes] [_ 'no]) → 'no

(struct-id pat ...) match struct-id instance，用于匹配struct-id所表示的struct，见于lexer-parser-eval

(list lvp ...) match sequence / vector of patterns

(vector lvp ...) 其中 lvp ::= pat | pat ooo

注意通过 a ... 匹配的多个值会组装成list

如 (match '(1 (2) (2) 5) [(list a b ... c) b] [_ 'else]) → '((2) (2))

pat ooo表示 greedily match pat instances

且有 ooo ::= ... | ..k | ___ | __k

zero or more / k or more

```
(match '(1 (2) (2) 5) [(list a b ... c) b] [_ 'else]) → '(2 2)
(match '(1 (2) (2) 5) [(list a _ ... b) (list a b)] [_ 'else]) → '(1 5)
```

#:when cond-expr 用于对patter matching进行辅助判断，使用的variable是在pat中绑定的

如 (match '(1 2 3) [(list a b c) #:when (= 6 (+ a b c)) 'yes] ['no]) → 'yes

(match '(1 (2) (2) 5) [(list a b ... c) #:when (empty? b) 'yes] ['no]) → 'no