EOPL3 Notes

# Inductive Sets of Data

# Data Abstraction

## Specifying Data via Interfaces

### 编程练习

#### Exercise 2.3 diff-tree

##### 树的构造

**zero表示:**(diff (one) (one))

zero = 1 - 1

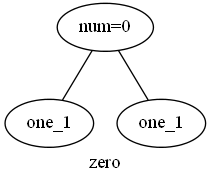


图 1 zero表示

**inc\_one/dec\_one的表示：**

|  |  |
| --- | --- |
| inc\_one = (diff (one) (diff (one) (one))) | dec\_one = (diff (diff (one) (one)) (one)) |
|  |  |

**子树相加：**设n(t)为两颗树相加的结果，现在求子树t1与t2如何组成t， n(t) = n(t1) + n(t2)

|  |  |  |
| --- | --- | --- |
| n(t) | = | n(t1) + n(t2) |
|  | = | n(t1) – (-n(t2)) |
|  | = | n(t1) – (0 – n(t2)) |
|  | = | n(t1) – ((1 - 1) – n(t2)) |

**diff-tree表示 =**

|  |
| --- |
| diff-tree = (diff **t1**  (diff (diff (one) (one))  **t2**)) |

**successor** = t + inc\_one

**predecessor** = t + dec\_one

successor与predecessor可以基于diff-tree加法实现

##### 接口编写

|  |
| --- |
| #lang eopl  (require "lib/eopl\_comm.rkt")  *;Diff-tree ::= (one) | (diff Diff-tree Diff-tree)*  (define one '(one))  (define inc-one '(diff (one) (diff (one) (one))))  (define dec-one '(diff (diff (one) (one)) (one)))  *;zero*  (define zero '(diff (one) (one)))  *;tree->num: Diff-tree -> Int*  (define (**tree->num** tree)  (if (equal? tree one) 1  (- (tree->num (lson tree))  (tree->num (rson tree)))))  *(display (tree->num inc-one)) (newline)*  *(display (tree->num dec-one)) (newline)*  ;is-zero?:Diff-tree -> boolean  (define (**is-zero?** tree)  (zero? (- (tree->num (lson tree))  (tree->num (rson tree)))))  *(display (is-zero? inc-one)) (newline)*  *(display (is-zero? dec-one)) (newline)*  ;diff-tree-plus:Diff-tree x Diff-tree -> Diff-Tree(n(t1) + n(t2))  ;usage:return n(t1) + n(t2) in constant time  (define (**diff-tree-plus** t1 t2)  (interior-node 'diff  t1  (interior-node 'diff  (interior-node 'diff one one)  t2)))  ;successor:Diff-tree -> Diff-tree(n + 1)  (define (**successor** tree) (diff-tree-plus tree inc-one))  *(display (tree->num (successor zero))) (newline)*  *(display (tree->num (successor one))) (newline)*  *(display (tree->num (successor inc-one))) (newline)*  *(display (tree->num (successor dec-one))) (newline)*  ;predecessor:Diff-tree -> Diff-tree(n - 1)  (define (**predecessor** tree) (diff-tree-plus tree dec-one))    *(display (tree->num (predecessor zero))) (newline)*  *(display (tree->num (predecessor one))) (newline)*  *(display (tree->num (predecessor inc-one))) (newline)*  *(display (tree->num (predecessor dec-one))) (newline)*  ;num->tree:Int(n) -> Diff-tree(n)  (define (**num->tree** n)  (if (zero? n) zero  (successor (num->tree (- n 1)))))  *(display (num->tree 3)) (newline)*  *(display (tree->num (num->tree 3))) (newline)*  ;diff-tree-plus test  (display (tree->num (diff-tree-plus (num->tree 10) (num->tree 20)))) (newline) |

## 数据类型的表示策略

**程序不依赖具体数据类型:** 当抽象数据类型被使用后，程序可以不依赖数据的具体表现形式。可以在不修改程序的情况下,重新定义抽象数据类型的部分接口实现.

### Environment接口

**environment定义:**变量的有限集合，变量可以是所有的Scheme值。

**表示形式：**{(var1, val1), . . ., (varn, valn)},

* vari是互不相同的变量, vali是任何Schem值。
* 称var的值与env绑定

**environment接口定义**

|  |  |  |
| --- | --- | --- |
| (empty-env) | = | ⌈∅⌉ |
| (apply-env *f*  *var*) | = | *f* (*var*) |
| (extend-env *var v* ⌈ *f* ⌉) | = | ⌈g⌉, 当*var1=var*时，*g(var1) = v；其它情况,g(var1)=f(var1)* |

**说明：**

* empty-env:没有参数，产生一个空的environment数据类型表示。
* (apply-env env var):查看env中var对应的值
* (extend-env *var val env*):生成一个新的env, var对应的值变更为val,没有var时新增。
* empty-env / extend-env为constructor，apply-env为observer

#### Exercise 2.4 stacks的接口定义

stack表示形式：{var1, var2, ... , varn}

|  |  |  |
| --- | --- | --- |
| (empty-stack) | = | ⌈∅⌉ |
| (push ⌈ *f* ⌉  *var*) | = | ⌈*g*⌉, ⌈*g*⌉ = {var, var1, var2, ... , varn} |
| (pop ⌈ *f* ⌉) | = | ⌈*g*⌉, ⌈*g*⌉ = {var2, var3, ... varn} |
| (top ⌈ *f* ⌉) | = | var1 |
| (empty-stack? ⌈ *f* ⌉) | = | ⌈ *f* ⌉ == ⌈∅⌉ |

### Environment数据结构的表示

**env的构造：**可以从empty-env开始，通过n次extend-env构成

|  |
| --- |
| (extend-env varn valn  ...  (extend-env var1 val1  (empty-env))...) |

从env的构造特点，提取出构造语法： empty-env/extend-env直接作为env的组成部分

|  |  |  |
| --- | --- | --- |
| Env-exp | ::= | (empty-env) |
|  | ::= | (extend-env Identifier Scheme-value Env-exp) |

Env-exp的语法与list语法是一样的，所以可以参考list来实现env

|  |
| --- |
| Env = (empty-env) | (extend-env Var SchemeVal Env) Var = Sym |

**apply-env：**如果env为空，报错。如果env由extend-env构造,则检查var是否在env中绑定。如果是，则返回env保存的值；否则继续在保存的env中寻找变量。

**iinterpreter模式：**按apply-env这种模式的代码

1. 检查一块数据
2. 判断它表示的是什么类型的数据
3. 抽取数据的组成部分，并进行合适的处理。

#### env的接口的实现

通过env的构造过程，直接转换为表示形式：

* empty: (empty-env)
* non-empty: (extend-env 'a 1 (extend-env 'b 2 (empty-env)))

**说明：**

var的惟一绑定和覆盖之前绑定值 这一特性，是指通过env接口表现出来的。而实现时env中可以有多个var对应不同的值，但只有满足取出最新的值就OK。

|  |
| --- |
| #lang eopl  (require racket/trace)  *;environment representation:*  *;empty: (empty-env)*  *;non-empty: (extend-env 'a 1 (extend-env 'b 2 (empty-env)))*  **;Grammer:**  **;Env-exp ::= (empty-env)**  **; ::= (extend-env Identifier Scheme-value Env-exp)**  *;report-no-binding-found: SchemeVal -> ErrorMsg*  (define (report-no-binding-found search-var)  (eopl:error 'apply-env "No Binding for ~s" search-var))  *;report-invalid-env: Env -> ErrorMsg*  (define (report-invalid-env env)  (eopl:error 'apply-env "Bad environment: ~s" env))  *; this way, define a variable not a function.*  *;(define empty-env '(empty-env))*  *;empty-env: () -> List; return '(empty-env)*  (define empty-env  (lambda () '(empty-env)))  (display (empty-env)) (newline)  *;empty-env?: Env -> boolean*  (define (empty-env? env)  (equal? env (empty-env)))  (display (empty-env? (empty-env))) (newline)  *;extend-env:Var × SchemeVal × Env → Env*  (define (extend-env var val env)  (list 'extend-env var val env))  *;apply-env:Env x Var -> SchemeVal*  (define (apply-env env search-var)  (cond ((null? env) (report-invalid-env env))  ((empty-env? env) (report-no-binding-found search-var))  ((not (equal? (car env) 'extend-env)) (report-invalid-env env))  ((not (eqv? 4 (length env))) (report-invalid-env env))  ((equal? search-var (cadr env)) (caddr env))  (else (apply-env (cadddr env) search-var))))  (display (apply-env '(extend-env a 1 (extend-env b 2 (empty-env)))  'b)) (newline) |

#### 编程练习

##### Exercise 2.6 01\_environment

**Grammar:**

|  |
| --- |
| Env ::= () | (env-element\*)  env-element ::= (Symbol SchemeVal) |

|  |
| --- |
| #lang eopl  *;environment representation:*  *;empty -> '()*  *;non-empty -> '(('a 1) ('b 2))*  *;empty-env: -> '()*  (define empty-env '())  *;extend-env: Var x SchemeVal x Env -> Env*  *;usage:return distinct var of Env*  (define (extend-env var val env)  (cond ((equal? env empty-env) (list (list var val)))  ((equal? var (caar env)) (cons (list (caar env) val) (cdr env)))  (else (cons (car env) (extend-env var val (cdr env))))))  (display (cons '(1 2) '((2 3)))) (newline)  (display (extend-env 'a 1 empty-env)) (newline)  (display (extend-env 'b 2 '((a 1)))) (newline)  (display (extend-env 'a 2 '((a 1)))) (newline)  *;apply-env: Env x Var -> SchemeVal*  (define (apply-env env search-var)  (cond ((null? env) (report-no-binding-found search-var))  ((not (= 2 (length (car env)))) (report-invalid-env env) )  ((equal? search-var (caar env)) (cadar env))  (else (apply-env (cdr env) search-var))))  *;(display (apply-env '() 'a)) (newline)*  (display (apply-env '((a 1)) 'a)) (newline)  (display (apply-env '((a 1) (a 2)) 'a)) (newline)  (display (apply-env '((a 1) (b 2)) 'b)) (newline) |

##### Exercise 2.6 02\_environment

Grammer: 在单层list中实现

|  |
| --- |
| env ::= () | ({var val}\*) |

**接口实现：**

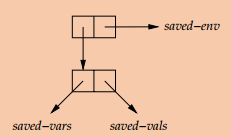
|  |
| --- |
| ;environment representation:  ;empty -> '()  ;non-empty -> '((a 1) (b 2))  ;empty-env: -> '()  (define empty-env (lambda () '()))  ;empty-env?: Env -> Boolean  (define (empty-env? env) (equal? env (empty-env)))  ;extend-env: Var x SchemeVal x Env -> Env  (define (extend-env var val env)  (append (list var val) env))  (display (extend-env 'a 1 (extend-env 'b 2 (empty-env)))) (newline)  ;apply-env: Env x SearchVar -> Val  (define (apply-env env search-var)  (cond ((empty-env? env) (report-no-binding-found search-var))  ((< (length env) 2) (report-invalid-env env))  ((equal? (car env) search-var) (cadr env))  (else (apply-env (cddr env) search-var))))  (display (apply-env '(a 1 b 2) 'b)) (newline) |

##### Exercise 2.6& 2.8 03\_environment

**Grammar:**

|  |
| --- |
| Env ::= () | (Env-element Env)  Env-element ::= (Symbol SchemeVal) |

示例：((a 1) ((b 2) ((c 3)))



**接口实现：**

|  |
| --- |
| ;environment representation:  *;empty -> '()*  *;non-empty -> '((a 1) ((b 2) ()))*  *;empty-env: -> '()*  (define empty-env (lambda () '()))  *;empty-env?: Env -> Boolean*  (define (empty-env? env) (equal? env (empty-env)))  *;extend-env: Var x SchemeVal x Env -> Env*  (define (extend-env var val env)  (list (list var val) env))  (display (extend-env 'a 1 (extend-env 'b 2 (empty-env)))) (newline)  *;apply-env: Env x SearchVar -> Val*  (define (apply-env env search-var)  (cond ((empty-env? env) (report-no-binding-found search-var))  ((< (length env) 2) (report-invalid-env env))  ((< (length (car env)) 2) (report-invalid-env env))  ((equal? (caar env) search-var) (cadar env))  (else (apply-env (cadr env) search-var))))  (display (apply-env '((a 1) ((b 2) ())) 'b )) (newline) |

## Interfaces for Recursive Data Types

#### zipper介绍

<http://blog.ezyang.com/2010/04/you-could-have-invented-zippers/>

##### list-zipper

list-zipper = (location (reversed-pre-list) (sub-list))，表示了整个list

**说明：**

* location也被称为focus点，该数据结构相当于C++中的相向链表。
* reversed-pre-list：用于向前构造list

**优点：**

* 具有数据不可变性质
* 方便在树的特点地点插入/删除值
* 适用于在list**当前节点进行插入、删除和前后移动**场景。比如：要重复在当前结构插入一批数据，在函数式语言中，就需要反复从头移到当前节点，然后构造新的list.如果list表示为list-zipper，就可以高效的进入插入操作。

**eg:**

[1, 2, 3, 4] 在location为3时（也称focus在3上），对应的list-zipper为([2, 1], [**3**, 4])。通过list-zipper可以重新构造出整个list

##### bintree-zipper

#### Exercise 2.18 NodeInSequence

**语法：**

|  |
| --- |
| NodeInSequence ::= (Int Left-Listof(Int) Right-Listof(Int2)) |

说明：

(1 2 3 4 5 6 7 8 9)使用(6 (5 4 3 2 1) (7 8 9))表示，以6为轴。左移/右移是相对于轴来说的。

##### 接口：

|  |  |  |
| --- | --- | --- |
| number->sequence | Int -> NodeInSequence | > (number->sequence 7) (7 () ()) |
| current-element | NodeInSequence –> Int | > (current-element ’(6 (5 4 3 2 1) (7 8 9))) 6 |
| move-to-left | Left-ListOf(Int)中的数据往左移，当前元素放到右边 | (move-to-left ’(6 (5 4 3 2 1) (7 8 9))) (5 (4 3 2 1) (6 7 8 9)) |
| move-to-right | Right-ListOf(int)中的数据往左移，当前元素放到左边 | > (move-to-right ’(6 (5 4 3 2 1) (7 8 9))) (7 (6 5 4 3 2 1) (8 9)) |
| insert-to-left | Int x NodeInSequence –> NodeInSequence  往Left-ListOf(Int)的开头插入 | > (insert-to-left 13 ’(6 (5 4 3 2 1) (7 8 9))) (6 (13 5 4 3 2 1) (7 8 9)) |
| insert-to-right | 往Right-ListOf(Int)的开头插入 | > (insert-to-right 13 ’(6 (5 4 3 2 1) (7 8 9))) (6 (5 4 3 2 1) (13 7 8 9)) |

##### 接口实现

|  |
| --- |
| #lang eopl  (require racket/pretty)  (require "../lib/eopl\_comm.rkt")  *;NodeInSequence ::= (Int Listof(Int) Listof(Int))*  *;(number->sequence 7) -> (7 () ())*  (define (number->sequence int)  `(,int () ()))  (pretty-print (number->sequence 7))  *;(current-element '(6 (5 4 3 2 1) (7 8 9))) -> 6*  (define (current-element node)  (car node))  (pretty-print (current-element '(6 (5 4 3 2 1) (7 8 9))))  *;(move-to-left ’(6 (5 4 3 2 1) (7 8 9)))*  *;->(5 (4 3 2 1) (6 7 8 9))*  (define (move-to-left node)  (if (null? (lson node))  (display "Out-of-Range")  (interior-node (car (lson node))  (cdr (lson node))  (cons (current-element node) (rson node)))))  (pretty-print (move-to-left '(7 () (8))))  (pretty-print (move-to-left '(6 (5 4 3 2 1) (7 8 9))))  *;(move-to-right '(6 (5 4 3 2 1) (7 8 9)))*  *;-> (7 (6 5 4 3 2 1) (8 9))*  (define (move-to-right node)  (if (null? (rson node))  (display "Out-of-Range")  (interior-node (car (rson node))  (cons (current-element node) (lson node))  (cdr (rson node)))))  (pretty-print (move-to-right '(6 (5 4 3 2 1) (7 8 9))))  *;(insert-to-left 13 '(6 (5 4 3 2 1) (7 8 9)))*  *;->(6 (13 5 4 3 2 1) (7 8 9))*  (define (insert-to-left int node)  (interior-node (current-element node)  (cons int (lson node))  (rson node)))  (pretty-print (insert-to-left 13 '(6 (5 4 3 2 1) (7 8 9))))  *;(insert-to-right 13 '(6 (5 4 3 2 1) (7 8 9)))*  *;->(6 (5 4 3 2 1) (13 7 8 9))*  (define (insert-to-right int node)  (interior-node (current-element node)  (lson node)  (cons int (rson node))))  (pretty-print (insert-to-right 13 '(6 (5 4 3 2 1) (7 8 9)))) |

#### Exercise 2.19 *Bintree*

**语法：**

|  |
| --- |
| Bintree ::= () | (Int Bintree Bintree) |

**接口：**

|  |  |  |
| --- | --- | --- |
| number->bintree | Int -> Bintree | (number->bintree 13) -> (13 () ()) |
| current-element | Bintree -> Int |  |
| move-to-left | 取左子树 |  |
| move-to-right | 取右子树 |  |
| at-leaf? | ()为叶节点 |  |
| insert-to-left | 插入左子树，作为原左子树的根 |  |
| insert-to-right | 插入右子树，作为原右子树的根 |  |

**接口实现：**使用racket的单元测框架**rackunit**

|  |
| --- |
| #lang eopl  **(require rackunit)**  (require "../lib/eopl\_comm.rkt")  (define (number->bintree int) `(,int () ()))  (define (current-element bintree) (contents-of bintree))  (define (move-to-left bintree) (lson bintree))  (define (move-to-right bintree) (rson bintree))  (define (at-leaf? bintree) (null? bintree))  (define (insert-to-left int bintree)  (interior-node (contents-of bintree)  (interior-node int  (lson bintree)  '())  (rson bintree)))  (define (insert-to-right int bintree)  (interior-node (contents-of bintree)  (lson bintree)  (interior-node int  (rson bintree)  '())))  (define t0 (number->bintree 13))  (define t1 (insert-to-right 14 (insert-to-left 12 t0)))  (check-equal? t0 '(13 () ()))  (check-equal? (current-element t1) 13)  (check-equal? t1 '(13 (12 () ()) (14 () ())))  (check-equal? (move-to-left t1) '(12 () ()))  (check-equal? (move-to-right t1) '(14 () ()))  (check-equal? (at-leaf? (move-to-right (move-to-left t1))) #t)  (check-equal? (insert-to-left 15 t1) '(13  (15  (12 () ())  ())  (14 () ()))) |

#### Exercise 2.20 bintree – zipper

https://wiki.haskell.org/Zipper

<http://learnyouahaskell.com/zippers>

http://chris-taylor.github.io/blog/2013/02/13/the-algebra-of-algebraic-data-types-part-iii/

##### 语法：

|  |
| --- |
| > data Tree a = Nil | Node a (Tree a) (Tree a)  > data Loc a = Loc (Tree a) (Context a)  > data Context a = Top  > | Left a (Tree a) (Context a)  > | Right a (Tree a) (Context a) |

##### 接口实现

|  |
| --- |
| #lang eopl  (require racket/trace)  (require rackunit)  (require racket/pretty)  (require "../lib/eopl\_comm.rkt")  ;> data Tree a = Nil | Node a (Tree a) (Tree a)  ;> data Loc a = Loc (Tree a) (Context a)  ;> data Context a = Top  ;> | Left a (Tree a) (Context a)  ;> | Right a (Tree a) (Context a)  ;---------------contructor-------------  (define (node-nil) '())  (define (leaf e) `(,e () ()))  (define (tree e left-tree right-tree)  `(,e ,left-tree ,right-tree))  (define (location tree ctxt) `(,tree ,ctxt))  (define (context-left e subtree pre-ctxt)  `(left ,e ,subtree ,pre-ctxt))  (define (context-right e subtree pre-ctxt)  `(right ,e ,subtree ,pre-ctxt))  (define (context-top) `(top))  ;---------------extractor--------------  (define (loc->tree loc) (car loc))  (define (loc->node loc) (contents-of (car loc)))  (define (loc->ltree loc) (lson (car loc)))  (define (loc->rtree loc) (rson (car loc)))  (define (loc->ctxt loc) (cadr loc))  (define (ctxt->dir ctxt) (car ctxt))  (define (ctxt->node ctxt) (cadr ctxt))  (define (ctxt->subtree ctxt) (caddr ctxt))  (define (ctxt->pre-ctxt ctxt) (cadddr ctxt))  ;---------------predicate---------------  (define (at-leaf? loc) (null? (loc->ltree loc)))  (define (at-root? loc) (equal? (loc->ctxt loc) (context-top)))  ;---------------bintree-zipper----------  (define (number->bintree int) (location (leaf int) (context-top)))  (define (current-element bintree) (loc->node bintree))  (define (move-to-left bintree)  (location (loc->ltree bintree)  (context-right (loc->node bintree)  (loc->rtree bintree)  (loc->ctxt bintree))))  (define (move-to-right bintree)  (location (loc->rtree bintree)  (context-left (loc->node bintree)  (loc->ltree bintree)  (loc->ctxt bintree))))  (define (insert-to-left int bintree)  (location (tree (loc->node bintree)  (tree int (loc->ltree bintree) (node-nil))  (loc->rtree bintree))  (loc->ctxt bintree)))  (define (insert-to-right int bintree)  (location (tree (loc->node bintree)  (loc->ltree bintree)  (tree int (loc->rtree bintree) (node-nil)))  (loc->ctxt bintree)))  (define (move-up bintree)  (let\* [(dir (ctxt->dir (loc->ctxt bintree)))  (child1 (loc->tree bintree))  (child2 (ctxt->subtree (loc->ctxt bintree)))  (parent (ctxt->node (loc->ctxt bintree)))  (pre-ctxt (ctxt->pre-ctxt (loc->ctxt bintree)))  (children (if (equal? 'left dir)  (list child2 child1)  (list child1 child2)))]  (location (tree parent (car children) (cadr children))  pre-ctxt)))  ;--------------bintree-zipper-test---------  (define top (number->bintree 12))  (pretty-print top)  (pretty-print (and (at-leaf? top) (at-root? top)))  (pretty-print (current-element top))  (define c1 (location (leaf 13) (context-left 27 (leaf 14) (context-top))))  (pretty-print c1)  (define c2 (insert-to-right 5 (insert-to-left 7 c1)))  (pretty-print c2) (pretty-print (current-element c2))  (define c3 (move-to-left c2))  (pretty-print c3)  (pretty-print (current-element c3))  (define c4 (move-to-right c2))  (pretty-print c4)  (define c5 (move-up c4)) (pretty-print c5)  (check-equal? c5 c2)  (define c6 (move-up (move-up c4))) (pretty-print c6)  (check-equal? (at-root? c6) #t) |

**输出结果：**

|  |
| --- |
| ((12 () ()) (top))  #t  12  ((13 () ()) (left 27 (14 () ()) (top)))  ((13 (7 () ()) (5 () ())) (left 27 (14 () ()) (top)))  13  ((7 () ()) (right 13 (5 () ()) (left 27 (14 () ()) (top))))  7  ((5 () ()) (left 13 (7 () ()) (left 27 (14 () ()) (top))))  ((13 (7 () ()) (5 () ())) (left 27 (14 () ()) (top)))  ((27 (14 () ()) (13 (7 () ()) (5 () ()))) (top)) |

##### move-to-left

|  |  |
| --- | --- |
| **ZipperTree(c2)**:  |->tree: Bintree:  |->val:**13**  |->ltree: Bintree:  |->val:7  |->ltree:null  |->rtree:null  |->rtree: Bintree:  |->val:**5**  |->ltree:null  |->rtree:null  |->ctxt: Ctxt:  |->dir:LEFT  |->val:27  |->subtree: Bintree:  |->val:14  |->ltree:null  |->rtree:null  |->ctxt: Ctxt:  |->dir:TOP  |->val:0  |->subtree:null  |->ctxt:null | **ZipperTree(c3)**:  |->tree: Bintree:  |->val:**7**  |->ltree:null  |->rtree:null  |->ctxt: Ctxt:  |->dir:**RIGHT**  |->val:**13**  |->subtree: Bintree:  |->val:**5**  |->ltree:null  |->rtree:null  |->ctxt: Ctxt:  |->dir:LEFT  |->val:27  |->subtree: Bintree:  |->val:14  |->ltree:null  |->rtree:null  |->ctxt: Ctxt:  |->dir:TOP  |->val:0  |->subtree:null  |->ctxt:null |

##### move-to-right

|  |  |
| --- | --- |
| **ZipperTree(c2)**:  |->tree: Bintree:  |->val:**13**  |->ltree: Bintree:  |->val:**7**  |->ltree:null  |->rtree:null  |->rtree: Bintree:  |->val:**5**  |->ltree:null  |->rtree:null  |->ctxt: Ctxt:  |->dir:LEFT  |->val:27  |->subtree: Bintree:  |->val:14  |->ltree:null  |->rtree:null  |->ctxt: Ctxt:  |->dir:TOP  |->val:0  |->subtree:null  |->ctxt:null | **ZipperTree(c4)**:  |->tree: Bintree:  |->val:**5**  |->ltree:null  |->rtree:null  |->ctxt: Ctxt:  |->dir:**LEFT**  |->val:**13**  |->subtree: Bintree:  |->val:**7**  |->ltree:null  |->rtree:null  |->ctxt: Ctxt:  |->dir:LEFT  |->val:27  |->subtree: Bintree:  |->val:14  |->ltree:null  |->rtree:null  |->ctxt: Ctxt:  |->dir:TOP  |->val:0  |->subtree:null  |->ctxt:null |

##### move-up

基于move-to-right的结果ZipperTree(c4)向上移动，恢复到ZipperTree(c2)。即ZipperTree(c2)= ZipperTree(c5)

|  |  |
| --- | --- |
| ZipperTree(c4):  |->tree: Bintree:  |->val:5  |->ltree:null  |->rtree:null  |->ctxt: Ctxt:  |->dir:LEFT  |->val:13  |->subtree: Bintree:  |->val:7  |->ltree:null  |->rtree:null  |->ctxt: Ctxt:  |->dir:LEFT  |->val:27  |->subtree: Bintree:  |->val:14  |->ltree:null  |->rtree:null  |->ctxt: Ctxt:  |->dir:TOP  |->val:0  |->subtree:null  |->ctxt:null | ZipperTree(c5):  |->tree: Bintree:  |->val:13  |->ltree: Bintree:  |->val:7  |->ltree:null  |->rtree:null  |->rtree: Bintree:  |->val:5  |->ltree:null  |->rtree:null  |->ctxt: Ctxt:  |->dir:LEFT  |->val:27  |->subtree: Bintree:  |->val:14  |->ltree:null  |->rtree:null  |->ctxt: Ctxt:  |->dir:TOP  |->val:0  |->subtree:null  |->ctxt:null |

**总结：**

move-to-left/right：代表向下移动，打开tree的子树，将未走过子树压缩到context中。

move-up：代表向上移动，打开context，利用 当前结点及其子树+父结点值+父结点另一子树 => 父结点树。并移动到父节点。

## 定义递归数据类型的工具

#### Exercise 2.28 unparse lc-exp to string

|  |
| --- |
| #lang eopl  (require rackunit)  (require racket/string)  **(define identifier? symbol?)**  **(define-datatype lc-exp lc-exp?**  (var-exp  (var identifier?))  (lambda-exp  (bound-var identifier?)  (body lc-exp?))  (app-exp  (rator lc-exp?)  (rand lc-exp?)))  **(define (unparse-exp e)**  (cases lc-exp e  (var-exp (var) (symbol->string var))  (lambda-exp (bound-var body)  (string-append "(lambda (" (symbol->string bound-var) ") "  (unparse-exp body) ")"))  (app-exp (rator rand)  (string-append "(" (unparse-exp rator) " "  (unparse-exp rand) ")"))))  *(define e (app-exp (lambda-exp 'a*  *(app-exp (var-exp 'a) (var-exp 'b)))*  *(var-exp 'c)))*  *(define e\_str "((lambda (a) (a b)) c)")*  *(check-equal? (unparse-exp e) e\_str)*  *(eopl:printf "All testcases passed!")* |

# Expressions

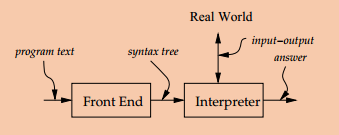
主要内容：变量绑定与作用域，使用environment保存每个变更和表达式的值

## Specification and Implementation Strategy

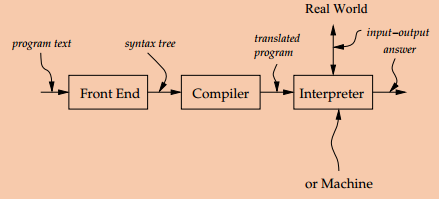
在environment ρ中exp的值应该是val,定义为：

|  |
| --- |
| (value-of *exp* ρ) = *val* |

* Program text：程序源代码，，由front-end转换为AST(abstract syntax tree)
* 解释器：AST -> Action
* implementation/defining language：编写解释器的语言



* target language：编译器将AST->其它（底层）语言，比如机器语言
* byte code & virtual machine : 编译器将源代码转换为相对简单的语言（byte code），并使用virtual machine解释执行byte code，从而屏蔽机器差异。



**编译器：**分为analyzer和translator两部分

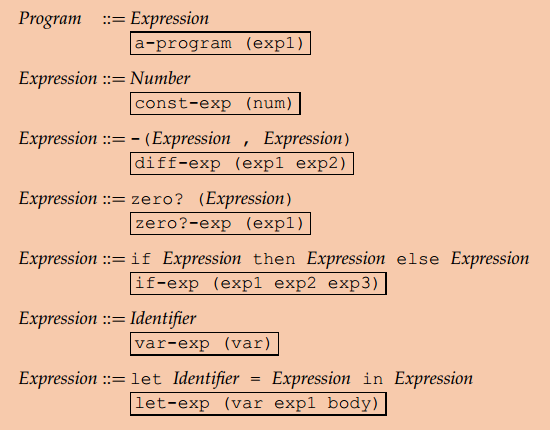
* scanning : Program Text -> Tokens, 由语言的lexical specification指定
* parsing : Tokens -> AST, 由语言的语法指定转换规则

**list表示程序：**

应用2.5节parse-expression的方法，使用list按结构表示数据，可避免scanning与parsing步骤,

## LET: A Simple Language

### LET语法



示例：LET程序

|  |
| --- |
| (scan&parse "-(55, -(x,11))") #(struct:a-program  #(struct:diff-exp  #(struct:const-exp 55)  #(struct:diff-exp  #(struct:var-exp x)  #(struct:const-exp 11)))) |

### Specification of Values

**expressed values** : 表达式所有可能的值

**denoted values** : 变量所有可能的值

LET的ExpVal和DenVal:

|  |
| --- |
| ExpVal = Int+ Bool  DenVal = Int+ Bool |

**ExpVal的接口：**

|  |  |
| --- | --- |
| **num-val** | : *Int*→ *ExpVal* |
| **bool-val** | : *Bool*→ *ExpVal* |
| **expval->num** | : *ExpVal*→ *Int* |
| **expval->bool** | : *ExpVal*→ *Bool* |

### Environments

参考前面的environment实现

**Environment抽象表示**：

|  |  |
| --- | --- |
| ρ | environments的值域范围 |
| [] | empty environment |
| [var = val]ρ | (extend-env var val ρ) |
| [var1 = val1, var2 = val2]ρ | [var1 = val1]([var2 = val2]ρ) |
| [var1 = val1, var2 = val2, . . .] | 在environment 中 var1的值为 val1, etc |

示例：

|  |  |
| --- | --- |
| [x=3]  [y=7]  [u=5]ρ | (extend-env ’x 3  (extend-env ’y 7  (extend-env ’u 5 ρ))) |

### 定义Expression的行为

根据LET语法结构抽象出如下接口：

**构造函数：**

|  |
| --- |
| ***const-exp*** *: Int→Exp*  ***zero?-exp*** *: Exp → Exp ;for boolean*  ***if-exp*** *: Exp × Exp × Exp → Exp*  ***diff-exp*** *: Exp × Exp → Exp*  ***var-exp*** *: Var → Exp*  ***let-exp*** *: Var × Exp × Exp → Exp* |

**observer** :

|  |
| --- |
| **value-of** : *Exp* × *Env* → *ExpVal* |

**接口的使用：**

|  |
| --- |
| (value-of (const-exp n) ρ) = (num-val n) ;常量的值是自身  (value-of (var-exp var) ρ) = (apply-env ρ var)  ; - (exp1, exp2)  (value-of (diff-exp exp1 exp2) ρ)  = (num-val  (-  (expval->num (value-of exp1 ρ))  (expval->num (value-of exp2 ρ)))) |

**表示法：**

* **«*exp*»** ：代表exp的AST
* ⌈n⌉ : 代表(num-val n)
* ⌊val⌋ : 代表(expval->num val)

推论：⌊⌈n⌉⌋ = n

LET程序仅是一个表达式，表达式的值通过指定 自由变量的值 确定。即：提供一个适合的初始化环境。eg:

|  |
| --- |
| (value-of-program exp)  = (value-of exp [i= 1, v= 5, x= 10]) |

#### LET表达式计算步骤示例

ρ = [i=1,v=5,x=10],计算-(-(x,3), -(v, i))表达式的值：

|  |
| --- |
| (value-of <<-(-(x,3), -(v, i))>> ρ))  =(num-val  (- (expval->num **(value-of <<-(x,3)>> ρ)**)  (expval->num (value-of <<-(v, i)>> ρ))))  =(num-val  (- **(expval->num**  **(- (expval->num (value-of <<x>> ρ)) ;x = 10**  **(expval->num (value-f <<3>> ρ))))**  (expval->num (value-of <<-(v, i)>> ρ))))  =(num-val  (- **7**  **(expval->num (value-of <<-(v, i)>> ρ))))**  =(num-val  (- **7**  **(expval->num**  **(- (exp->val (value-of <<v>> ρ))**  **(exp->val (value-f <<i>> ρ))))))**  =(num-val (- 7 **(- 5 1)**))  =(num-val **(- 7 4)**)  **=(num-val 3)** |

### Specifying Conditionals

**语法：**

*Expression* ::= if *Expression* then *Expression* else *Expression*

**应用：**

**(value-of (zero?-exp *exp*1) ρ)**

= (bool-val #t), if (expval->num val1) = 0

= (bool-val #f), if (expval->num val1) != 0

**(value-of (if-exp *exp*1 *exp*2 *exp*3) ρ)**

= (value-of exp2 ρ), if (zero?-exp exp1) = (bool-val #t)

= (value-of exp3 ρ), if (zero?-exp exp1) = (bool-val #f)

**if-exp计算规则：**

|  |
| --- |
| **(value-of (if-exp exp1 exp2 exp3) ρ)**  = (if (expval->bool (value-of exp1 ρ))  (value-of exp2 ρ)  (value-of exp3 ρ)) |

if-exp计算示例：ρ = [x= 33,y= 22]

|  |
| --- |
| (value-of <<if zero?(-(x,11)) then -(y,2) else -(y,4)>> ρ)  =(if (expval->bool **(value-of << zero?(-(x,11))>> ρ)**)  (value-of <<-(y,2)>> ρ)  (value-of <<-(y,4)>> ρ))  =(if **(expval->bool (bool-val #f)) ;** ⌊⌈n⌉⌋ = n  (value-of <<-(y,2)>> ρ)  (value-of <<-(y,4)>> ρ))  =(if #f  (value-of <<-(y,2)>> ρ)  **(value-of <<-(y,4)>> ρ)**)  =(value-of <<-(y,4)>> ρ)  =(num-val 18) |

### Specifying let

**语法：**

*Expression* ::= let *Identifier* = *Expression* in *Expression*

**let表达式:**

|  |
| --- |
| let x = 5  in -(x,3) |

表示：在作用域in中，x的值为5

**规则：**

|  |
| --- |
| (value-of exp1 ρ) = val1 |
| (value-of (let-exp var exp1 body) ρ) = (value-of body [var = val1]ρ) |

**let计算规则：**

|  |
| --- |
| (value-of (let-exp var exp1 body) ρ) = (value-of body [var=(value-of exp1 ρ)]ρ) |

### LET的实现

附录

# Mathematcial Symbol

向上取整：⌈...⌉

向下取整⌊⌋

# graphViz

参考：<http://graphs.grevian.org/example>

## 无向图

|  |  |
| --- | --- |
| graph {  rankdir=LR  a -- b;  a -- c;  a -- e;    b -- c;  e -- c;    c -- d;  } |  |

**说明：**

* **--：** 无向图连线，可指定 节点集合 到 节点集合的连线，实现批量指定连线。
* **rankdir=LR:**指定图形排布方式，**TB:**从上到下, **BT:**从下到上; **LR:**从左到右; **RL:**从右到左。默认为TB
* **位置：**图形和连线出现的位置与图形元素的声明顺序有关

## 带标签加权无向图

|  |
| --- |
| graph {  rankdir=LR  a [label="start",**color**=red,**style**=filled]  d [label="end",**color**=black,**fontcolor**=white,**style**=filled]  a -- b[**label**="10",**weight**="10"];  a -- c[label="1",weight="1"];  a -- e[label="1",weight="1"];    b -- c[label="4",weight="4"];  e -- c[label="1",weight="1"];    c -- d[label="6",weight="6"];  } |
|  |

**说明：**

* label=”start”:指定节点或连线的标签内容
* color=red:指定节点的颜色
* fontcolor=white:指定节点标签字体的颜色
* style=filled:指定 节点或连线 风格，filled:填充满, 对于线:style=dotted:虚线
* weight:两个节点之间的连线weight越大，则节点靠得更近。

## 有向图

|  |  |
| --- | --- |
| digraph {  a -> b[label="0.2",weight="2"];  a -> c[label="0.4",weight="4"];  c -> b[label="0.6",weight="6"];  c -> e[label="0.6",weight="6"];  e -> e[label="0.1",weight="1"];  e -> b[label="0.7",weight="7"];  } |  |

**说明：**

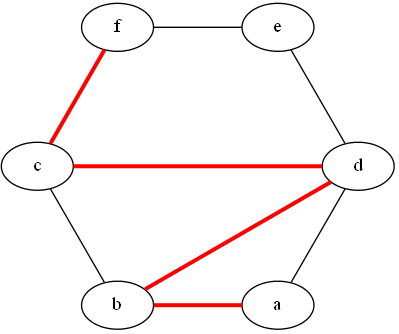
**画图引擎：**要画出右图，需要切换画图引擎为circo.

### 画图引擎

|  |  |
| --- | --- |
| 图形引擎 | 特点 |
| circo | 适合多环路结构的图形 |
| dot |  |
| neato |  |
| fdp |  |
| sfdp |  |
| twopi | 放射状布局 |

## 标注路径

由circo图形引擎生成：



|  |
| --- |
| graph {  a -- b -- d -- c -- f[color=red,penwidth=3.0];  b -- c;  d -- e;  e -- f;  a -- d;  } |

## 子图

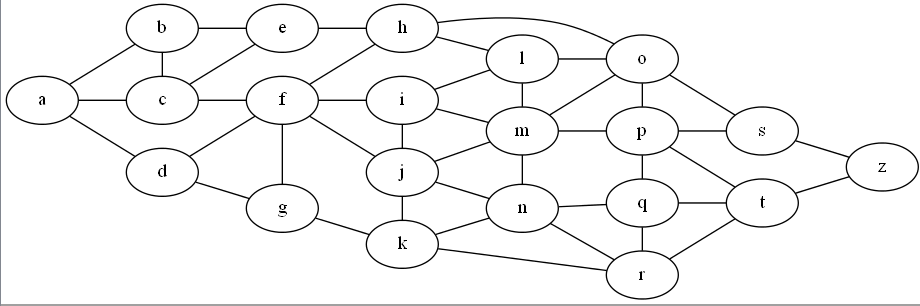
由dot图形引擎生成：

|  |  |
| --- | --- |
| digraph {  **splines=line;**  subgraph **cluster**\_0 {  label="Subgraph A";  a -> b;  b -> c;  c -> d;  }  subgraph cluster\_1 {  label="Subgraph B";  a -> f;  f -> c;  }  } |  |

**说明：**

* splines=line：指定只使用直线
* subgraph clusterXXX: 子图命名必需以cluster开关，否则无法合并到一个框图中。而且只有dot引擎支持。

## 大型图形:rank=same对齐



|  |
| --- |
| graph {  rankdir=LR;  a -- { b c d }; b -- { c e }; c -- { e f }; d -- { f g }; e -- h;  f -- { h i j g }; g -- k; h -- { o l }; i -- { l m j }; j -- { m n k };  k -- { n r }; l -- { o m }; m -- { o p n }; n -- { q r };  o -- { s p }; p -- { s t q }; q -- { t r }; r -- t; s -- z; t -- z;  { rank=same b c d };  { rank=same e f g } ;  { rank=same h i j k };  { rank=same l m n };  { rank=same o p q r };  { rank=same s t };  } |

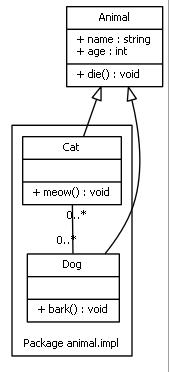
**说明：**

* a -- { b c d }; ： 指定 节点 -> 节点集合 的连线
* rank=same：将节点对齐排列，左右或上下
* ranksep=1：指定两级rank之间的距离,inch. 在上图中ranksep越大，a,c离的越开
* nodesep=1: 指定同级rank之间的距离,inch. 在上图中nodesep越大,a,c,d离的越开

## UML元素

引用：http://www.ffnn.nl/pages/articles/media/uml-diagrams-using-graphviz-dot.php

使用dot引擎生成：如果使用circo引擎，则无法生成子图



|  |
| --- |
| digraph G {  fontname = "Bitstream Vera Sans"  fontsize = 8  node [  fontname = "Bitstream Vera Sans"  fontsize = 8  shape = "record"  ]  edge [  fontname = "Bitstream Vera Sans"  fontsize = 8  ]  Animal [  label = "{Animal|+ name : string\l+ age : int\l|+ die() : void\l}"  ]  subgraph **cluster**AnimalImpl {  label = "Package animal.impl"  Dog [  label = "{Dog||+ bark() : void\l}"  ]  Cat [  label = "{Cat||+ meow() : void\l}"  ]  }  edge [  arrowhead = "empty"  ]  Dog -> Animal  Cat -> Animal  edge [  arrowhead = "none"  headlabel = "0..\*"  taillabel = "0..\*"  ]  Dog -> Cat  } |

### node[..],edge[..]设置节点、连线属性

### node[shape=”record”]

设置节点为record，这样的节点可以被分割，适合构造类图

### 类表示：Animal Class

|  |
| --- |
| Animal [  label = "{Animal|+ name : string\l+ age : int\l|+ die() : void\l}"  ] |
| Animal class UML model |

**说明：**

* "{" and "}"：表示要创建一个record的图形，并带有分隔线。
* "|" ： 代表分隔线。这时用于分隔类名、方法、属性
* "\l" ： 换行，后面的字符左对齐

### 继承关系:edge[arrowhead = "empty"]

|  |  |
| --- | --- |
| edge [  arrowhead = "empty"  ]  Dog -> Animal  Cat -> Animal | Adding the subclass relations to the UML diagram |

### N:M关系:edge[arrowhead=”none”,headlabel=””,taillabel=””]

|  |  |
| --- | --- |
| edge [  arrowhead = "none"  headlabel = "0..\*"  taillabel = "0..\*"  ] | Adding the association between classes |

### 包：使用子图实现subgraph clusterxxx {}

|  |  |
| --- | --- |
| subgraph clusterAnimalImpl {  label = "Package animal.impl"  ... Cat/Dog类  } |  |