Lab 1: CSG 711: Programming to Structure

Karl Lieberherr
Using Dr. Scheme

• context-sensitive help
• use F1 as your mouse is on an identifier. HelpDesk is language sensitive. Be patient.
• try the stepper
• develop programs incrementally
• definition and use: Check Syntax
• use 299 / intermediate with lambda
General Recipe

• Write down the requirements for a function in a suitable mathematical notation.
• Structural design recipe: page 368/369
HtDP
Designing Algorithms

• Data analysis and design
• Contract, purpose header
• Function examples
• Template
• Definition
  – what is a trivially solvable problem?
  – what is a corresponding solution?
  – how do we generate new problems
  – need to combine solutions of subproblems
• Test
Template

(define (generative-rec-fun problem)
  (cond
    [(trivially-solvable? problem)
      (determine-solution problem)]
    [else
      (combine-solutions … problem …
        (generative-rec-fun (gen-pr-1 problem))
        …
        (generative-rec-fun (gen-pr-n problem)))]))
Template for list-processing

(define (generative-rec-fun problem)
  (cond
    [(empty? problem) (determine-solution problem)]
    [else
      (combine-solutions
        problem
        (generative-rec-fun (rest problem)))]))
duple (EOPL page 24)

(duple n x)
li:= empty;
for i :=1 to n do add x to li (does not matter where);

Structural recursion:
if i=0 empty
else (cons x (duple (- n 1)))
History (Programming to Structure)

• Frege: Begriffsschrift 1879: “The meaning of a phrase is a function of the meanings of its immediate constituents.”

• Example:
AppleList : Mycons | Myempty.
Mycons = <first> Apple <rest> AppleList.
Apple = <weight> int.
Myempty = .
Meaning of a list of apples?

Total weight

- \((t\text{Weight}\ al)\)
  - \([(\text{Myempty}\ ?\ al)\ 0]\)
  - \([(\text{Mycons}\ ?\ al)\]
    
    \((\text{Apple-weight}(\text{Mycons-first}\ al))\)
    
    \(\text{// meaning of first constituent}\)
    
    +
    
    \((t\text{Weight}(\text{Mycons-rest}\ al))\)]
    
    \(\text{// meaning of rest constituent}\)

AppleList : Mycons | Myempty.
Mycons = <first> Apple <rest> AppleList.
Apple = <weight> int.
Myempty = .

PL independent
In Scheme: Structure

(define-struct Mycons (first rest))
(define-struct Apple (weight))
(define-struct Myempty ())
Design Information

AppleList : Mycons | Myempty.
Mycons = <first> Apple <rest> AppleList.
Apple = <weight> int.
Myempty = .

(define-struct Mycons (first rest))
(define-struct Apple (weight))
(define-struct Myempty ())
In Scheme: Behavior

(define (tWeight al)
  (cond
   [(Myempty? al) 0]
   [(Mycons? al) (+
                 (Apple-weight (Mycons-first al))
                 (tWeight (Mycons-rest al))))])
In Scheme: Testing

(define list1 (make-Mycons (make-Apple 111) (make-Myempty)))

(tWeight list1)
111

(define list2 (make-Mycons (make-Apple 50) list1))

(tWeight list2) 161

Note: A test should return a Boolean value.

See tutorial by Alex Friedman on testing in DrabScheme.
Reflection on Scheme solution

• Program follows structure
• Design translated somewhat elegantly into program.
• Dynamic programming language style.
• But the solution has problems!
Behavior

• While the purpose of this lab is programming to structure, the Scheme solution uses too much structure!

```scheme
(define (tWeight al)
  (cond
    [(Myempty? al) 0]
    [(Mycons? al) (+
                   (Apple-weight (Mycons-first al))
                   (tWeight (Mycons-rest al)))]))
```

duplicates all of it!
How can we reduce the duplication of structure?

• First small step: Express all of structure in programming language once.
• Eliminate conditional!
• Implementation of tWeight() has a method for Mycons and Myempty.
• Extensible by addition not modification.
• Big win of OO.
Solution in Java

AppleList: abstract int tWeight();
Mycons: int tWeight() {
    return (first.tWeight() + rest.tWeight());
}
Myempty: int tWeight() {return 0;}

Translated to Java
What is better?

- structure-shyness has improved.
- No longer enumerate alternatives in functions.
- Better follow principle of single point of control (of structure).
Problem to think about (while you do hw 1)

• Consider the following two Shape definitions.
  – in the first, a combination consists of exactly two shapes.
  – in the other, a combination consists of zero or more shapes.

• Is it possible to write a program that works correctly for both shape definitions?
First Shape

Shape : Rectangle | Circle | Combination.
Rectangle = "rectangle" <x> int <y> int <width> int <height> int.
Circle = "circle" <x> int <y> int <radius> int.
Combination = "(" <top> Shape <bottom> Shape ")".
Second Shape

Shape : Rectangle | Circle | Combination.
Rectangle = "rectangle" <x> int <y> int <width> int <height> int.
Circle = "circle" <x> int <y> int <radius> int.
Combination = "(" List(Shape) ")".
List(S) ~ {S}.
Input (for both Shapes)

(  
  rectangle 1 2 3 4  
  (  
    circle 3 2 1  
    rectangle 4 3 2 1  
  )  
)
Think of a shape as a list!

- A shape is a list of rectangles and circles.
- Visit the elements of the list to solve the area, inside and bounding box problems.
Help with the at function

• Design the function at. It consumes a set S and a relation R. Its purpose is to collect all the seconds from all tuples in R whose first is a member of S.
Deriving Scheme solution (1)

at: s: Set r: Relation
Set s0 = {};
    from r:Relation to p:Pair:
        if (p.first in s) s0.add(p.second);
return s0;

definition

at: s: Set r: Relation
if empty(r) return empty set else {
    Set s0 = {}; p1 := r.first();
    if (p1.first in s) s0.add(p1.second);
    return union(s0, at(s, rest(r)))

decompose based on structure of a relation: either it is empty or has a first element
Deriving Scheme solution (2)

at: s: Set r: Relation
Set s0 = {};
  from r: Relation to p: Pair:
    if (p.first in s) s0.add(p.second);
return s0;

at: s: Set r: Relation
if empty(r) return empty set else {
  p1 := r.first(); rst = at(s, rest(r));
  if (p1.first in s) return rst.add(p1.second) else rst
}

Why not implement this definition directly using iteration ???

decompose based on structure of a relation:
either it is empty or has a first element
Close to final solution

;; at : Symbol Relation -> Set
(define (at s R)  at: s: Set r: Relation
  (cond         if empty(r) return empty set else {
    [(empty? R) empty-set]    p1 := r.first(); rst = at(s, rest(r));
      if (p1.first in s)  return rst.add(p.second)
    [else (local ((define p1 (first R)))
        else rst}      else rst
      (define rst (at s (rest R))))
      (if (element-of (first p1) s)
        (define (add-element (second p1) rst)
          (add-element (second p1) rst)))])
)
dot example

• Compute the composition of two relations.
• $r$ and $s$ are relations. $r.s$ (dot $r$ $s$) is the relation $t$ such that $x \, t \, z$ holds iff there exists a $y$ so that $x \, r \, y$ and $y \, s \, z$. 
Why not implement iterative solution?

dot Relation r1, r2
Relation r0 = {}; 
from r1: Relation to p1: Pair 
    from r2: Relation to p2: Pair 
        if (= p1.second p2.first) r0.add( new Pair(p1.first,p2.second));
    return r0;

if empty(r1) return empty-set else ;; there must be a first element p11 in r1 
    Relation r0 = empty-set; 
    from r2: Relation to p2: Pair 
        if (= p11.second p2.first) r0.add(new Pair(p11.first,p2.second));
    return union (r0, dot((rest r1),r2));
Closer to Scheme solution: reuse at

dot Relation r, s;
if empty(r) return empty-set else
;; there must be a first element fst in r
  x=fst.first; y=fst.second;
  zs = at(list(y), s);
turn x and zs into list of pairs: r0;
return union (r0, dot((rest r),s));
Scheme solution

(define (dot.v0 r s)
  (cond
    [(empty? r) empty]
    [else (local ((define fst (first r))
                   (define x (first fst))
                   (define y (second fst))
                   (define zs (at (list y) s)))
      (union (map (lambda (s) (list x s)) zs)
             (dot.v0 (rest r) s))])))
Save for later
Abstractions

• abstraction through parameterization:
  – planned modification points

• aspect-oriented abstractions:
  – unplanned extension points
Structure

• The Scheme program has lost information that was available at design time.
  – The first line is missing in structure definition.
  – Scheme allows us to put anything into the fields.

AppleList : Mycons | Myempty.
Mycons = <first> Apple <rest> AppleList.
Apple = <weight> int.
Myempty = .
Information can be expressed in Scheme

- Dynamic tests
- Using object system