Lab 1: CSG 711: Programming to Structure

Karl Lieberherr
History

• 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

\[ (tWeight\ al) \]
- \[(Myempty?\ al)\ 0\]
- \[(Mycons?\ al)\]
  \[
  (Apple-weight(Mycons-first\ al))
  // meaning of first constituent
  \]
  +
  \[
  (tWeight(Mycons-rest\ al)))
  // meaning of rest constituent
  \]

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 ())
(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 list1)
161
Reflection on Scheme solution

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

• The Scheme program has lost information that was available at design time.
  – The first line is missing.
  – 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
Behavior

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

```
(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

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

• abstraction through parameterization:
  – planned modification points
• aspect-oriented abstractions:
  – unplanned extension points