6.001 SICP
Tagged data

• Why do we need tags
• Concept of tags
• Extended example

Manipulating complex numbers

Complex number has:
- Real, imag, mag, angle

Addition easier in Cartesian coordinates

(define (+c z1 z2)
  (make-complex-from-rect (+ (real z1) (real z2))
   (+ (imag z1) (imag z2))))

Multiplication easier in polar coordinates

(define (*c z1 z2)
  (make-complex-from-polar (* (mag z1) (mag z2))
   (+ (angle z1) (angle z2))))

Bert's data structure

(define (make-complex-from-rect rl im)
  (list rl im))

(define (make-complex-from-polar mg an)
  (list (* mg (cos an))
       (* mg (sin an))))

(define (real cx) (car cx))
(define (imag cx) (cadr cx))
(define (mag cx) (sqrt (+ (square (real cx))
                      (square (imag cx)))))
(define (angle cx) (atan (imag cx) (real cx)))

Ernie's data structure

(define (make-complex-from-rect rl im)
  (list (sqrt (+ (square rl) (square im)))
        (atan im rl)))

(define (make-complex-from-polar mg an)
  (list mg an))

(define (real cx) (* (mag cx) (cos (angle cx))))
(define (imag cx) (* (mag cx) (sin (angle cx))))
(define (mag cx) (car cx))
(define (angle cx) (cadr cx))

Whose number is it?

• Suppose we pick up the following object

  What number does this represent?

Labeled complex numbers

(define (make-complex-from-rect rl im)
  (list 'rect rl im))

(define (make-complex-from-polar mg an)
  (list 'polar mg an))

(define (real sz) (cond ((eq? (tag obj) 'rect) (car (contents obj))
                        (eq? (tag obj) 'polar) (* (car (contents obj))
                         (cos (cadr (contents obj))))))

(define (imag sz) (cond ((eq? (tag obj) 'rect) (cadr (contents obj))
                        (eq? (tag obj) 'polar) (* (car (contents obj))
                         (sin (cadr (contents obj))))))

(define (tag obj) (car obj))
(define (contents obj) (cdr obj))

(define (real sz)
  (cond ((eq? (tag obj) 'rect) (car (contents obj)))
        (eq? (tag obj) 'polar) (* (car (contents obj))
         (cos (cadr (contents obj))))))

(define (imag sz)
  (cond ((eq? (tag obj) 'rect) (cadr (contents obj)))
        (eq? (tag obj) 'polar) (* (car (contents obj))
         (sin (cadr (contents obj)))))))

(define (angle sz)
  (cond ((eq? (tag obj) 'rect) (cadr (contents obj)))
        (eq? (tag obj) 'polar) (* (car (contents obj))
         (cos (cadr (contents obj)))))))
The concept of a tag

• Tagged data
  • attach an identifying symbol to all nontrivial data values
  • always check the symbol before operating on the data

(define (make-point x y) (list 'point x y))

Benefits of tagged data

• data-directed programming:
  functions that decide what to do based on the arguments
  • example: in a graphics program
    area: triangle|square|circle -> number

• defensive programming:
  functions that fail gracefully if given bad arguments
  -- much better to give an error message than
to return garbage!

Example: Arithmetic evaluation

(define exp1 (make-sum (make-sum 3 15) 20))
exp1                ==> (+ (+ 3 15) 20)
(eval-1 exp1)       ==> 38

Expressions might include values other than numbers
Ranges:
some unknown number between min and max
arithmetic: [3,7] + [1,3] = [4,10]
Limited precision values:
some value±some error amount
arithmetic: (100 ± 1) + (3 ± 0.5) = (103 ± 1.5)

Approach: start simple, then extend

• Characteristic of all software engineering projects
• Start with eval for numbers, then add support for ranges and limited-precision values
• Goal: build eval in a way that it will extend easily & safely
  • Easily: requires data-directed programming
  • Safely: requires defensive programming

• Today: multiple versions of eval
eval-1 Simple arithmetic, no tags
eval-2 Extend the evaluator, observe bugs
eval-3 through -7 Do it again with tagged data

1. ADT (Abstract Data Type) for sums

; type: Exp, Exp -> SumExp
(define (make-sum addend augend)
  (list '+ addend augend))

; type: anytype -> boolean
(define (sum-exp? e)
  (and (pair? e) (eq? (car e) '+)))

; type: SumExp -> Exp
(define (sum-addend sum) (cadr sum))
(define (sum-augend sum) (caddr sum))

• Type Exp will be different in different versions of eval

1. Eval for numbers only

; type: number | SumExp -> number
(define (eval-1 exp)
  (cond
    ; base case
    ; recursive case
    (else
      (error "unknown expression " exp))))

(eval-1 (make-sum 4 (make-sum 3 5))) ==> 12
2. ADT for ranges (no tags)

; type: number, number -> range2
(define (make-range-2 min max) (list min max))

; type: range2 -> number
(define (range-min-2 range) (car range))
(define (range-max-2 range) (cadr range))

; type: range2, range2 -> range2
(define (range-add-2 r1 r2)
  (make-range-2
   (+ (range-min-2 r1) (range-min-2 r2))
   (+ (range-max-2 r1) (range-max-2 r2))))

2. Eval for numbers and ranges (broken)

; type: number | range2 | SumExp -> number | range2
(define (eval-2 exp)
  (cond
   ((number? exp) exp)
   ((sum-exp? exp)
    (let ((v1 (eval-2 (sum-addend exp)))
          (v2 (eval-2 (sum-augend exp))))
     (if (and (number? v1) (number? v2))
         (+ v1 v2) ; add numbers
         (range-add-2 v1 v2))) ; add ranges
    ((pair? exp) exp) ; a range
    (else (error "unknown expression " exp))))

2. Ways in which eval-2 is broken

• Missing a case: sum of number and a range
  (eval-2 (make-sum 4 (make-range-2 4 6)))
  ==> error: the object 4 is not a pair

2. Eval for numbers and ranges (broken)

; type: number | range2 | SumExp -> number | range2
(define (eval-2 exp)
  (cond
   ((number? exp) exp)
   ((sum-exp? exp)
    (let ((v1 (eval-2 (sum-addend exp)))
          (v2 (eval-2 (sum-augend exp))))
     (if (and (number? v1) (number? v2))
         (+ v1 v2) ; add numbers
         (range-add-2 v1 v2))) ; add ranges
    ((pair? exp) exp) ; a range
    (else (error "unknown expression " exp))))
2. Ways in which eval-2 is broken

• Missing a case: sum of number and a range
  (eval-2 (make-sum 4 (make-range-2 4 6)))
  ==> error: the object 4 is not a pair

• Not defensive: what if we add limited-precision values but forget to change eval-2?
  (define (make-limited-precision-2 val err)
   (list val err))
  (eval-2 (make-sum
           (make-range-2 4 6)
           (make-limited-precision-2 10 1)))
  ==> (14 7)  correct answer: (13 17) or (15 2)

2. Lessons from eval-2

• Common bug: calling a function on the wrong type of data
  • typos
  • brainos
  • changing one part of the program and not another

• Common result: the function returns garbage
  • Why? Prim. predicates (number?, pair?) are ambiguous
  • Something fails later, but cause is hard to track down
  • Worst case: program produces incorrect output!!

• Next: how to use tagged data to ensure the program halts immediately

3. Start again using tagged data

• Take another look at SumExp ... it’s already tagged!
  (define sum-tag '+)
  ; Type: Exp, Exp -> SumExp
  (define (make-sum addend augend)
   (list sum-tag addend augend))
  ; Type: anytype -> boolean
  (define (sum-exp? e)
   (and (pair? e) (eq? (car e) sum-tag)))

• sum-exp? is not ambiguous: only true for things made by make-sum (assuming the tag + isn’t used anywhere else)

3. An ADT for numbers using tags

(define constant-tag 'const)
  ; type: number -> ConstantExp
  (define (make-constant val)
   (list constant-tag val))
  ; type: anytype -> boolean
  (define (constant-exp? e)
   (and (pair? e) (eq? (car e) constant-tag)))
  ; type: ConstantExp -> number
  (define (constant-val const) (cadr const))

3. Eval for numbers with tags (incomplete)

; type: ConstantExp | SumExp -> number
(define (eval-3 exp)
  (cond
   ((constant-exp? exp) (constant-val exp))
   ((sum-exp? exp)
    (+ (eval-3 (sum-addend exp))
       (eval-3 (sum-augend exp))))
   (else (error "unknown expr type: " exp)))

(eval-3 (make-sum (make-constant 3)
                  (make-constant 5))) ==> 8

• Not all nontrivial values used in this code are tagged

4. Eval for numbers with tags

; type: ConstantExp | SumExp -> ConstantExp
(define (eval-4 exp)
  (cond
   ((constant-exp? exp) exp)
   ((sum-exp? exp)
    (make-constant
     (+ (eval-4 (sum-addend exp))
        (eval-4 (sum-augend exp))))
    )
   (else (error "unknown expr type: " exp)))))

(eval-4 (make-sum (make-constant 3)
                  (make-constant 5)))
  ==> (constant 8)

There is that pattern of using selectors to get parts, doing something, then using constructor to reassemble
4. Make `add` an operation in the Constant ADT

```
(define (constant-add c1 c2)
  (make-constant (+ (constant-val c1) (constant-val c2))))
```

4. Lessons from `eval-3` and `eval-4`
- Standard pattern for an ADT with tagged data
- A variable in the ADT implementation stores the tag
- Attach the tag in the constructor
- Write a predicate that checks the tag
- Determines whether an object belongs to the ADT
- Operations strip the tags, operate, attach the tag again
- Must use tagged data everywhere to get full benefits
- Including return values

5. Same pattern: range ADT with tags

```
(define range-tag 'range)
(define (make-range min max)
  (list range-tag min max))
(define (range-exp? e)
  (and (pair? e) (eq? (car e) range-tag)))
(define (range-min range) (cadr range))
(define (range-max range) (caddr range))
```

5. Eval for numbers and ranges with tags

```
(define (eval-5 exp)
  (cond
   ((constant-exp? exp) exp)
   ((range-exp? exp) exp)
   ((sum-exp? exp)
    (let ((v1 (eval-5 (sum-addend exp)))
         (v2 (eval-5 (sum-augend exp))))
     (if (and (constant-exp? v1) (constant-exp? v2))
         (constant-add v1 v2)
         (range-add (val2range v1) (val2range v2))))
   (else (error "unknown expr type: " exp)))))
```

6. Simplify eval with a data-directed add function

```
(define value-exp? v)
  (or (constant-exp? v) (range-exp? v)))
(define (value-add-6 v1 v2)
  (if (and (constant-exp? v1) (constant-exp? v2))
      (constant-add v1 v2)
      (range-add (val2range v1) (val2range v2))))
```

6. Coercion to turn constants into ranges

```
(define (val2range val)
  (if (range-exp? val)
      val
      (make-range (constant-val val) (constant-val val))))
```

This is called coercion
6. Simplified eval for numbers and ranges

\[ \text{ValueExp = ConstantExp | RangeExp} \]
\[ \text{type: ValueExp | SumExp -> ValueExp} \]

\( \text{(define (eval-6 exp)} \)
\( \text{  (cond} \)
\( \text{    ((value-exp? exp) exp)} \)
\( \text{    ((sum-exp? exp)} \)
\( \text{      (value-add-6 (eval-6 (sum-addend exp)})} \)
\( \text{      (eval-6 (sum-augend exp))))} \)
\( \text{  (else (error "unknown expr type: " exp)))} \)

7. Eval for all data types

\( \text{(define limited-tag 'limited)} \)
\( \text{(define (make-limited-precision val err)} \)
\( \text{  (list limited-tag val err))} \)
\( \text{(define (eval-7 exp)} \)
\( \text{  (cond} \)
\( \text{    ((value-exp? exp) exp)} \)
\( \text{    ((limited-exp? exp)} \)
\( \text{      (value-add-6 (eval-7 (sum-addend exp)})} \)
\( \text{      (eval-7 (sum-augend exp))))} \)
\( \text{    (else (error "unknown expr type: " exp)))} \)

7. value-add-6 is not defensive

\( \text{(eval-7 (make-sum)} \)
\( \text{  (make-range 4 6)} \)
\( \text{  (make-limited-precision 10 1)))} \)

\( \text{(define (value-add-6 v1 v2)} \)
\( \text{  (if (and (constant-exp? v1) (constant-exp? v2))} \)
\( \text{    (constant-add v1 v2)} \)
\( \text{    (range-add (val2range v1) (val2range v2))))} \)

\( \text{• Correct answer should have been (range 13 17) or (limited 15 2)} \)

7. value-add-6 is not defensive

\( \text{(define (value-add-6 v1 v2)} \)
\( \text{  (if (and (constant-exp? v1) (constant-exp? v2))} \)
\( \text{    (constant-add v1 v2)} \)
\( \text{    (range-add (val2range v1) (val2range v2))))} \)

\( \text{(define (val2range val)} \)
\( \text{  (if (range-exp? val)} \)
\( \text{    val)} \)
\( \text{    (make-range (constant-val val)) ; assumes constant (constant-val val))))} \)
7. Defensive: check tags before operating

```
define (value-add-7 v1 v2)
  (cond
    ((and (constant-exp? v1) (constant-exp? v2))
      (constant-add v1 v2))
    ((and (value-exp? v1) (value-exp? v2))
      (range-add (val2range v1) (val2range v2)))
    (else
      (error "unknown exp: " v1 " or " v2))))
```

- Rule of thumb: when checking types, use the else branch only for errors

---

7. Lessons from eval-5 through eval-7

- Data directed programming can simplify higher level code
- Using tagged data is only defensive programming if you check the tags
  - don’t use the else branch of if or cond
- Traditionally, ADT operations and accessors don’t check tags
  - Omitted for efficiency; assume checked at the higher level
  - A check in constant-val would have trapped this bug
  - Add checks into your ADT implementation to be paranoid
- Andy Grove: “only the paranoid survive”