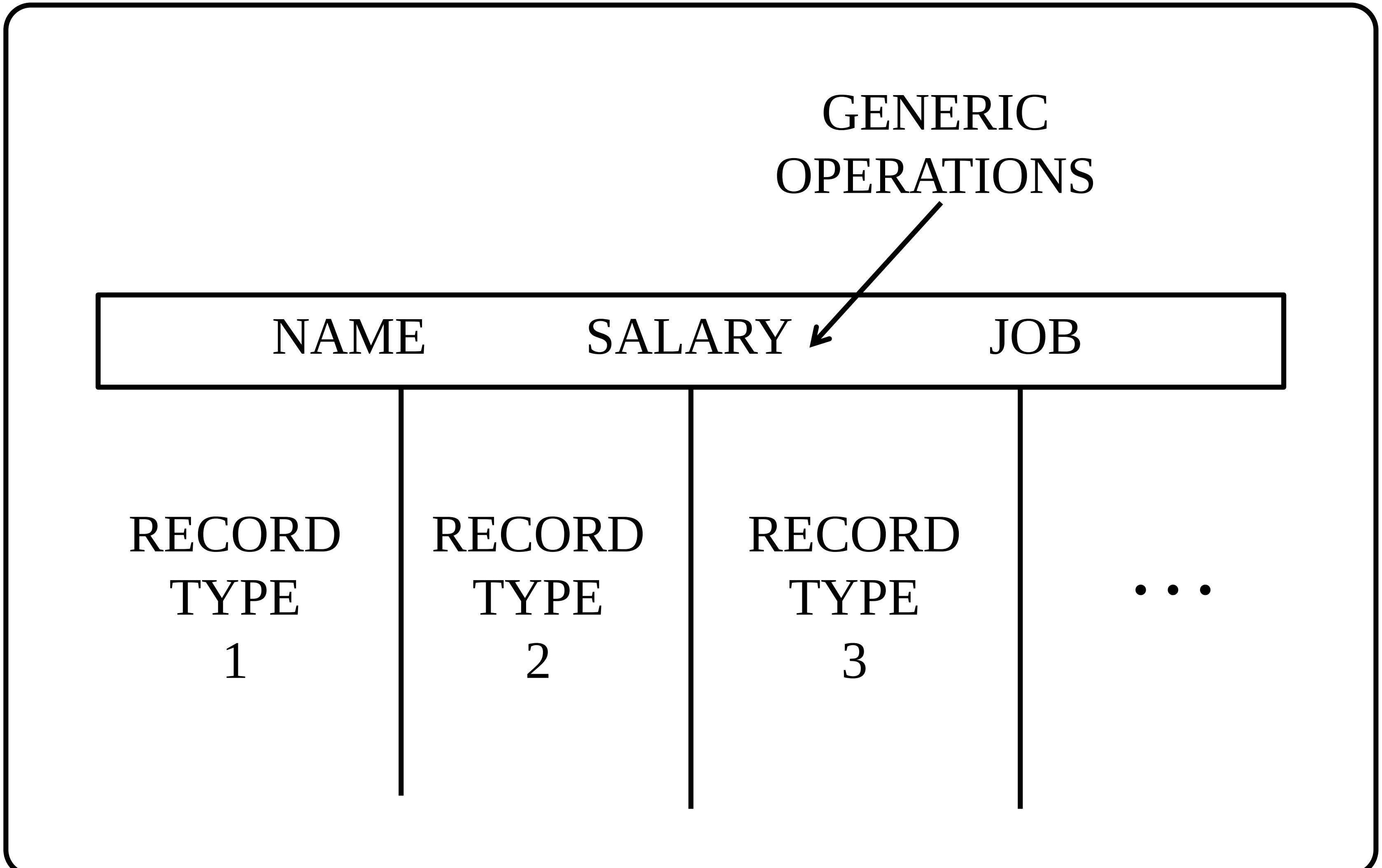
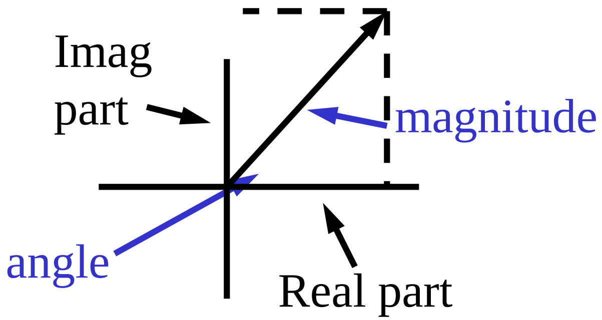
Tagged Data

- Tag: a symbol in a data structure that identifies its type
- Why we need tags
- Extended example: evaluating arithmetic expressions

Generic Operations



Manipulating complex numbers



Complex number has:

- real and imaginary part (Cartesian)
- magnitude and angle (polar)

Addition is easier in Cartesian coordinates

```
(define (+c z1 z2)
   (make-rectangular (+ (real-part z1)(real-part z2))
                     (+ (imag-part z1)(imag-part z2))))
```

Multiplication is easier in polar coordinates

```
(define (*c z1 z2)
   (make-polar
            (* (magnitude z1) (magnitude z2))
            (+ (angle z1) (angle z2))))
```

Bert's data structure

```
(define (make-rectangular  rl  im) (cons rl im))
(define (make-polar  mg  an)
    (cons (* mg (cos an))
          (* mg (sin an))))
```

Note conversion to rectangular form before storing

```
(define (real-part cx) (car cx))
(define (imag-part cx) (cdr cx))
(define (magnitude cx)
        (sqrt (+ (square (real cx))
                 (square (imag cx)))))
(define (angle cx) (atan (imag cx) (real cx)))
```

Need to do some computation since stored in rectangular form

Ernie's data structure

```
(define (make-rectangular rl im)
  (cons (sqrt (+ (square rl) (square im)))
          (atan im rl)))
(define (make--polar mg an) (cons mg an))
```

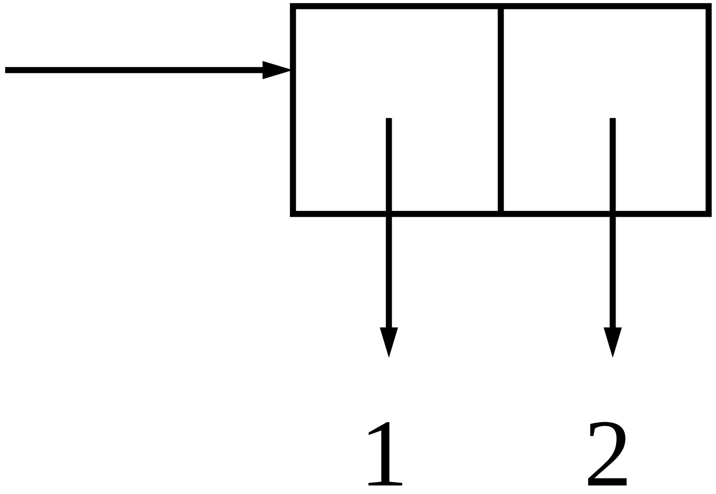
Note conversion to polar form before storing

```
(define (real-part cx)
          (* (mag cx) (cos (angle cx))))
(define (imag-part cx)
          (* (mag cx) (sin (angle cx))))
(define (magnitude cx) (car cx))
(define (angle cx) (cdr cx))
```

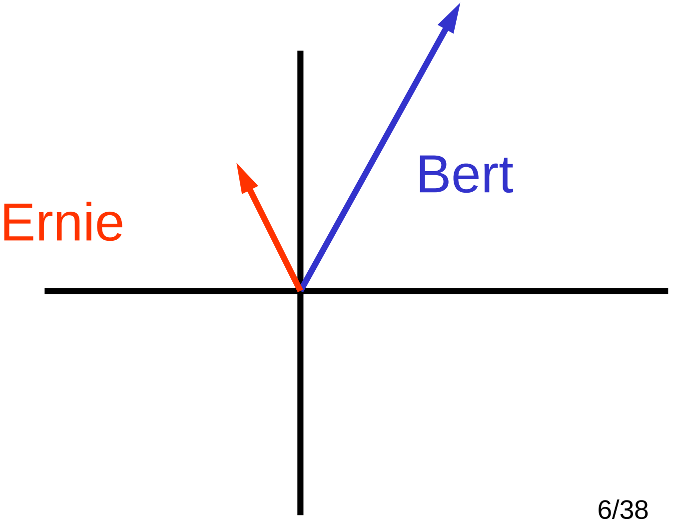
Need to do some computation since stored in polar form

Whose number is it?

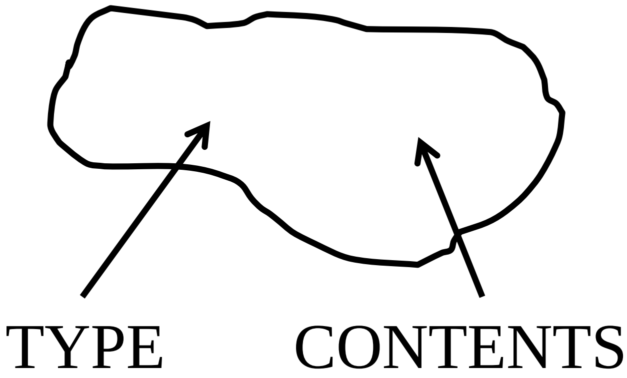
- Suppose we pick up the following object



- What number does this represent?



Typed Data




```
(define (attach-type  type  contents)
        (cons  type  contents))
(define (type  datum)
        (car  datum))
(define (contents  datum)
        (cdr  datum))
```

```
(define (rectangular?  z)
        (eq?  (type  z)  ‘rectangular))
(define (polar?  z)
        (eq?  (type z)  ‘polar))
```

Rectangular Package

```
(define (make-rectangular  x  y)
        (attach-type  ‘rectangular  (cons x y)))
(define (real-part-rectangular  z)   (car z))
(define (imag-part-rectangular z)  (cdr z))

(define (magnitude-rectangular  z)
        (sqrt (+ (square (car z))
                 (square (cdr z)))))

(define (angle-rectangular  z)
        (atan  (cdr z)  (car z)))
```

Polar Package

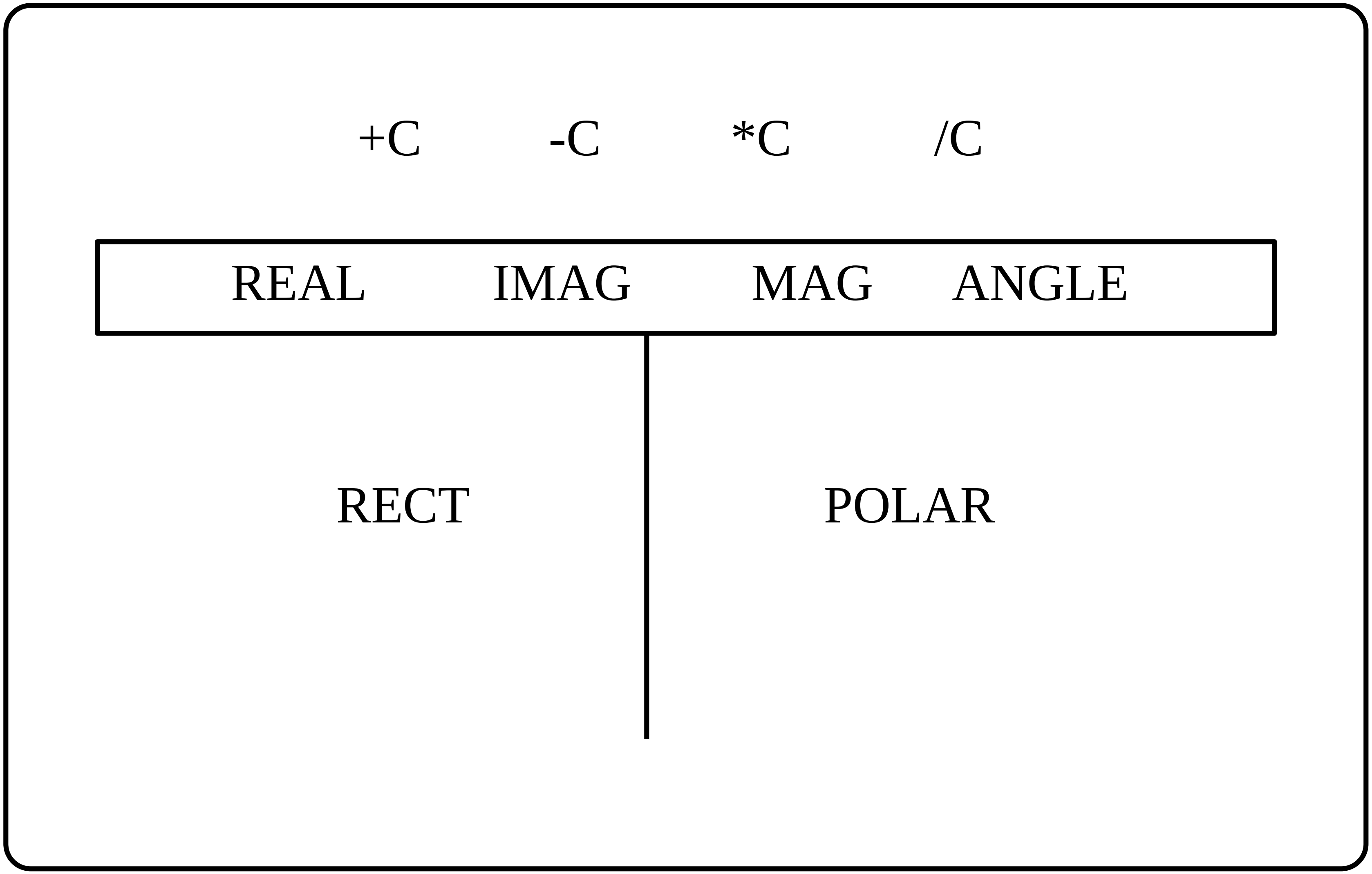
```
(define (make-polar  r  a)
        (attach-type  ‘polar  (cons r  a))
(define (real-part-polar  z)
        (*  (car z)  (cos (cdr z))))
(define (imag-part-polar z)
        (*  (car z)  (sin (cdr  z))))

(define (magnitude-polar   z)   (car   z))
(define (angle-polar  z)  (cdr  z))
```

Generic Selection from Complex Numbers

```
(define (REAL-PART  z)
  (cond  ((rectangular?  z)   … )
         ((polar?  z)  … )))
(define (IMAG-PART z)
  (cond ((rectangular?  z)   … )
        ((polar?  z)  … )))
(define (MAGNITUDE  z)
  (cond ((rectangular?  z)   … )
        ((polar?  z)  … )))
(define (ANGLE  z)
  (cond ((rectangular?  z)   … )
        ((polar?  z)  … )))
```

```
(define (real-part  z)
  (cond ((rectangular? z)
         (real-part-rectangular
           (contents  z)))
        ((polar? z)
         (real-part-polar
           (contents z)))))
  …
```

Complex Number



Operation Table

	POLAR	RECT
REAL-PART	REAL-PART-POLAR	REAL-PART-RECT
IMAG-PART	IMAG-PART-POLAR	IMAG-PART-RECT
MAGNITUDE	MAGNITUDE-POLAR	MAGNITUDE-RECT
ANGLE	ANGLE-POLAR	ANGLE-RECT

```
(PUT  KEY1  KEY2  VALUE)
(GET  KEY1  KEY2)
```

Install Operations in the table

- Installing the rectangular operations in the table

  (put ‘rectangular ‘real-part real-part-rectangular)

  (put ‘rectangular ‘imag-part imag-part-rectangular)

  (put ‘rectangular ‘magnitude magnitude-rectangular)

  (put ‘rectangular ‘angle angle-rectangular)

- Installing the polar operations in the table

  (put ‘polar ‘real-part real-part-polar)

  (put ‘polar ‘imag-part imag-part-polar)

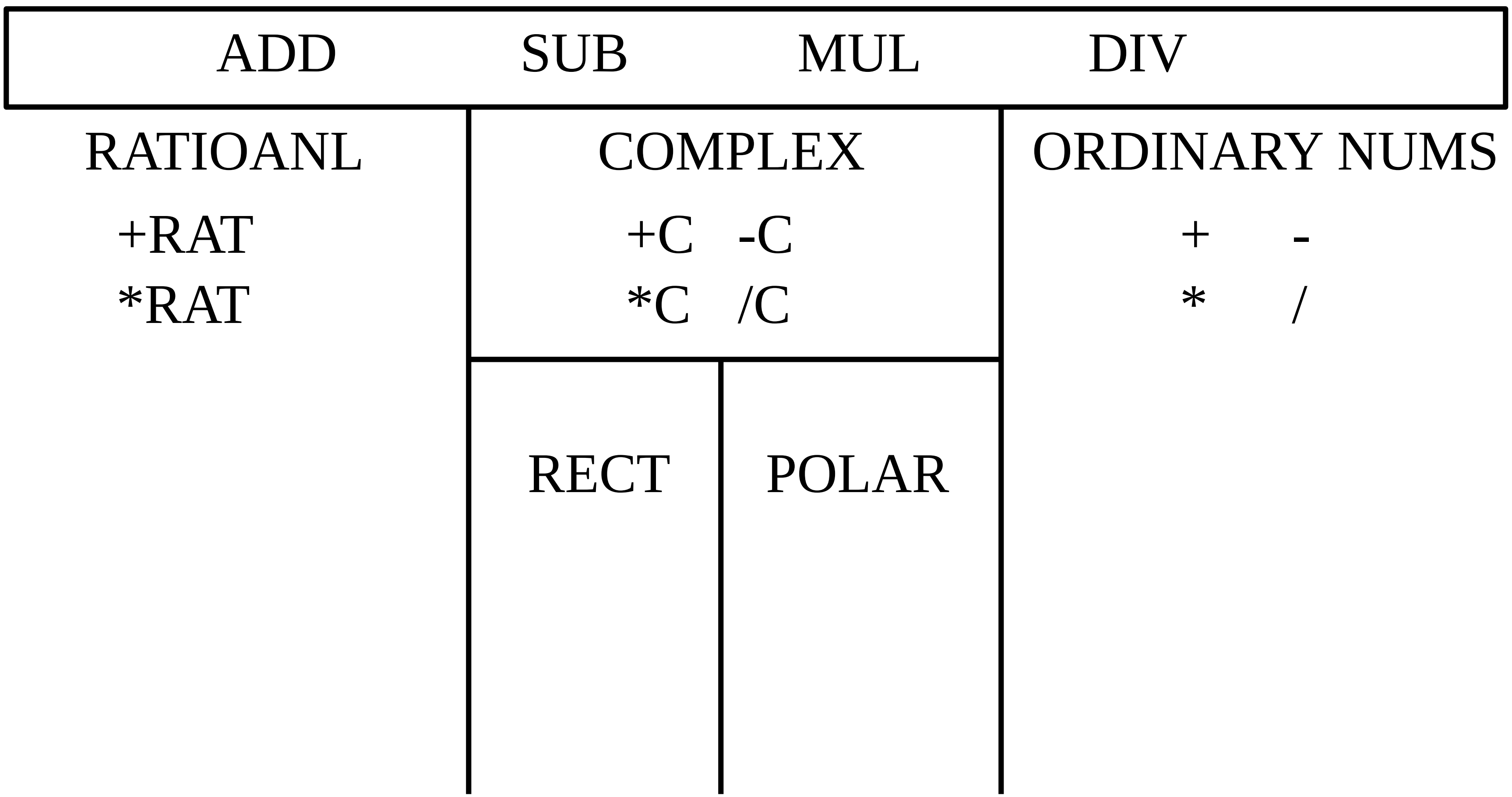
  (put ‘polar ‘magnitude magnitude-polar)

  (put ‘polar ‘angle angle-polar)

Dispatch on Type – Data Directed Programming

```
(define (operate  op  obj)
        (let  ((proc  (get (type obj)  op)))
            (if  (not  (null?  proc))
                (proc  (contents  obj))
                (error  "undefined operation"))))

(define (real-part obj)  (operate  'real-part  obj))
(define (imag-part obj)  (operate  'imag-part  obj))
(define (magnitude obj)  (operate  'magnitude  obj))
(define (angle obj)  (operate 'angle  obj))
```

Generic Arithmetic System



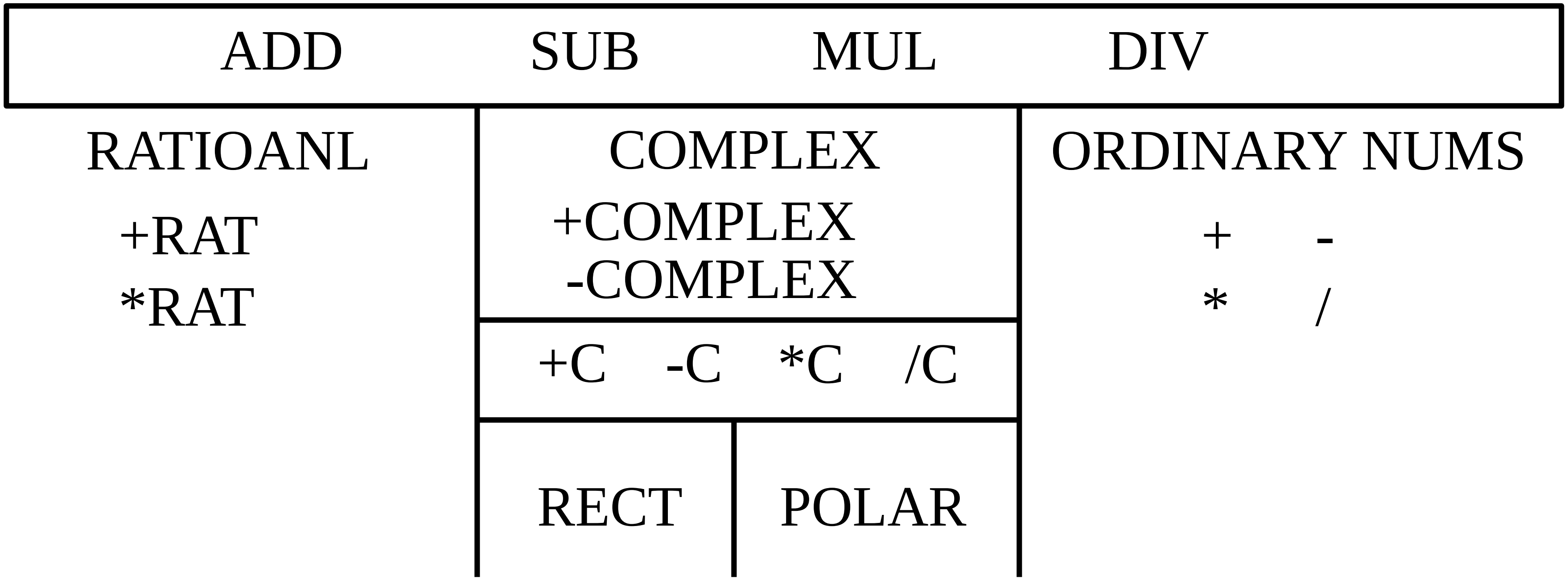
Rational Number

```
(define  (+ rat   x   y)
        (make-rat  (+  (*  (numer  x)  (denom  y))
                           (*  (denom x)  (numer  y)))
                       (*  (denom  x)   (denom  y))))
(define  (-rat  x   y)     …)
(define  (*rat  x   y)     …)
(define  (/rat   x   y)     …)

(define (make-rat  x   y)   (attach-type  ‘rational  (cons x y)))
(put  ‘rational  ‘add  +rat)
(put   ‘rational  ‘sub  -rat)
(put   ‘rational  ‘mul  *rat)
(put   ‘rational  ‘div   /rat)
```

operate-2

```
(define  (ADD   x   y)
        (OPERATE-2   'ADD    x   y))

(define  (operate-2  op   arg1   arg2)
  (if   (eq?  (type  arg1)   (type arg2))
      (let   ((proc   (get (type  arg1)   op)))
          (if   (not   (null?   proc))
              (proc  (contents  arg1)
                     (contents  arg2))
              (error  "op: undefined on type")))
      (error   "args not same type")))
```

Installing Complex Number



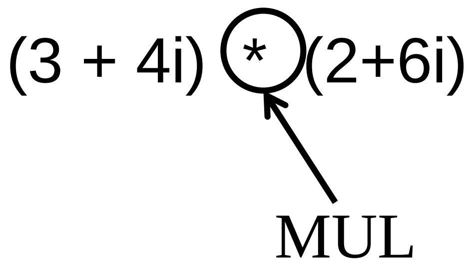
(define (make-complex z) (attach-type ‘complex z))

(define (+complex z1 z2) (make-complex (+c z1 z2)))

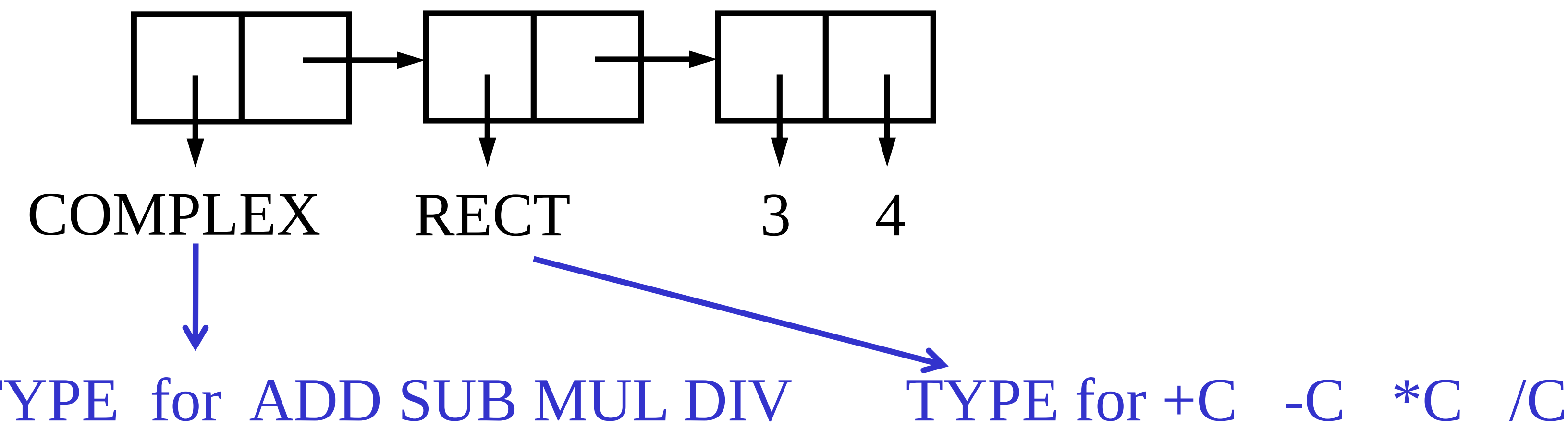
(put ‘complex ‘add +complex)

similarly for -complex, *complex, /complex

Complex Number Example




```
(define (MUL x y)
    (OPERATE-2 'MUL x y))
```



Installing Polynomials

$X^{15} + 2X^{7} + 5$ ➔ ((15 1) (7 2) (0 5))

(POLYNOMIAL X <TERM-LIST>)

```
(define (make-polynomial  var  term-list)
   (attach-type  'polynomial  (cons  var  term-list)))
(define (+poly  p1  p2)
   (if  (same-var?  (var p1)  (var p2))
        (make-polynomial   (var p1)   (+terms (term-list p1)
                                              (term-list  p2)))
        (error   "Polys  not  in same var")))

(put   'polynomial   'add  +poly)
```

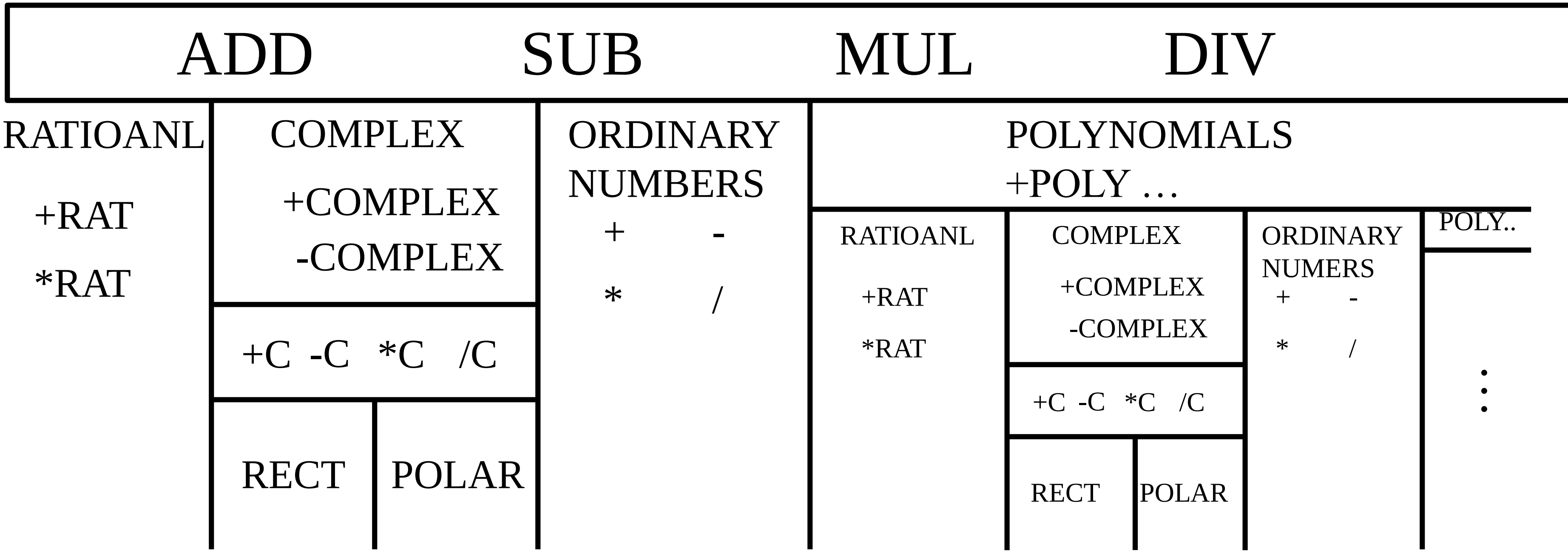
Installing Polynomials

```
(define (+terms L1 L2)
  (cond ((empty-termlist? L1)  L2)
        ((empty-termlist? L2)  L1)
        (else
          (let  ((t1 (first-term L1))
                 (t2 (first-term L2)))
            (cond
                 ((> (order t1) (order t2))  … )
                 ((< (order t1) (order t2))  … )
                 (else  … )))))))
```

Installing Polynomials

```
((>  (order   t1)   (order  t2)
  (adjoin-term  t1   (+terms  (rest-terms  L1)   L2)))
((<  (order  t1)   (order  t2)
  (adjoin-term   t2  (+term  L1  (rest-terms  L2))))
(else  (adjoin-terms
                    (make-term  (order  t1)
                                        (ADD  (coeff  t1)   (coeff  t2)))
                    (+terms  (rest-terms L1)  (rest-terms L2))))
```

Generic Arithmetic System




```
(define (+ rat x y)
        (make-rat (ADD (MUL (numer x) (denom y))
                           (MUL (denom x) (numer y)))
                      (MUL (denom x) (denom y))))
```

Benefits of tagged data

- data-directed programming:
  functions that decide what to do based on argument types

  - 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

Create arithmetic expressions

```
(define an-expr (make-sum (make-sum 3 15) 20))
an-expr           ==> (+ (+ 3 15) 20)
(eval an-expr)    ==> 38
```

Evaluate arithmetic expressions to reduce them to simpler form

Expressions might include values other than simple 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

- Process: 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. Data abstraction for sums of numbers

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

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

(define (sum-addend sum) (second sum))
(define (sum-augend sum) (third sum))
     ; type: SumExp -> Exp
```

- the type Exp will be different in different versions of eval

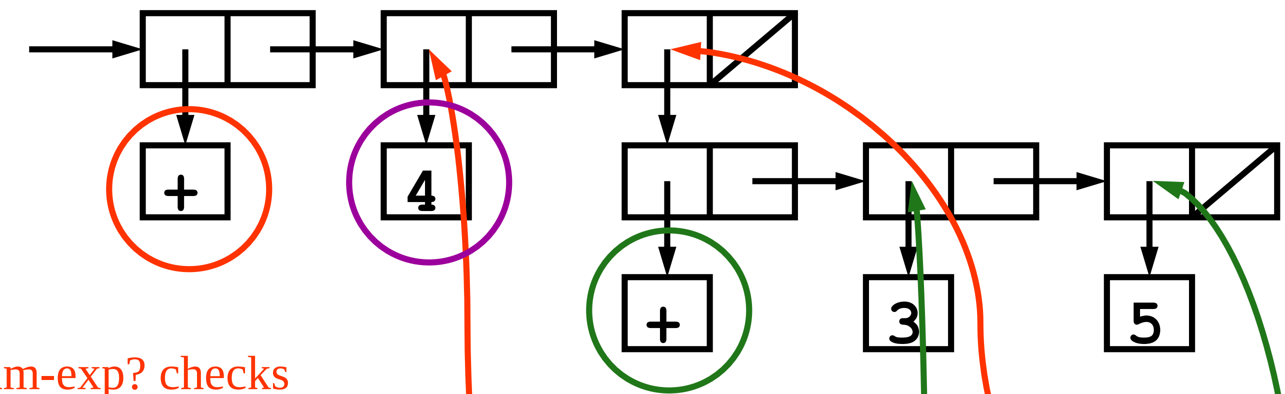
1. Eval for sums of numbers

```
; Exp = number | SumExp
(define (eval-1 exp)
   ; type: Exp -> number
   (cond
      ((number? exp)  exp)            ; base case
      ((sum-exp? exp)            ; recursive case
           (+ (eval-1 (sum-addend exp))
              (eval-1 (sum-augend exp))))
      (else
         (error "unknown expression " exp))))
```

```
(eval-1 (make-sum 4 (make-sum 3 5))) ==> 12
```

Example in gory detail

`(eval-1 (make-sum 4 (make-sum 3 5))) ==> 12`



Sum-exp? checks this using eq?

`(+ (eval-1 . ) (eval-1 . ))`

Number? checks this

Sum-exp? checks this using eq?

`(+ 4 (+ (eval-1 . ) (eval-1 . )))`

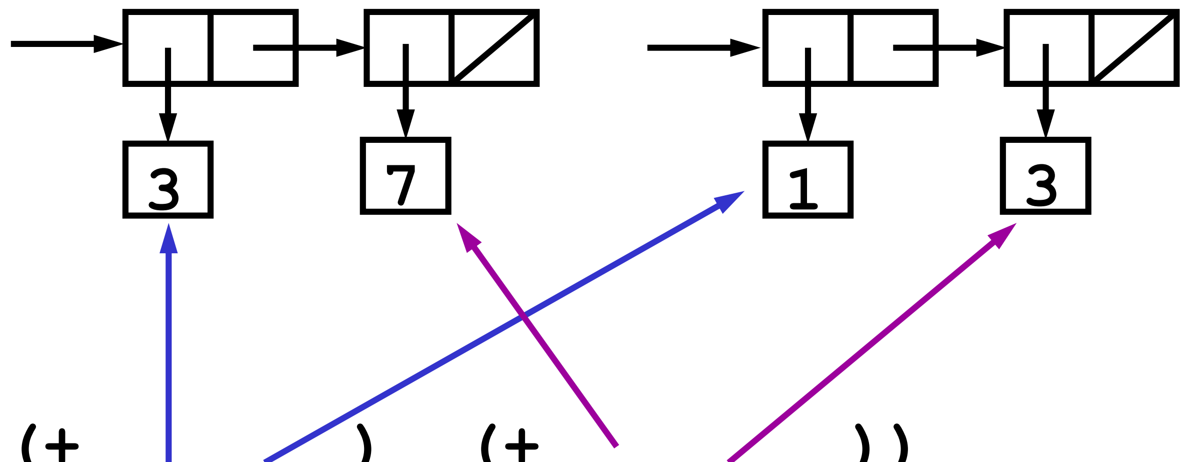
`(+ 4 (+ 3 5))`

2. Extend the abstraction to ranges (without tags)

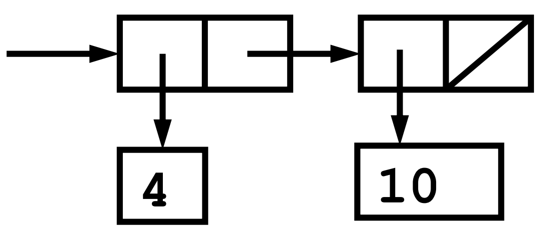
```
; type: number, number -> range2
(define (make-range-2 min max) (list min max))

; type: range2 -> number
(define (range-min-2 range) (first range))
(define (range-max-2 range) (second 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))))
```

Detailed example of adding ranges

```
(range-add-2 (make-range 3 7) (make-range 1 3))
```




```
(make-range-2 (+      )   (+          ))
(make-range-2 4 10)
```



This is a range

2. Eval for sums of numbers and ranges (broken!)

```
; Exp = number | range2 | SumExp
(define (eval-2 exp)
  ; type: Exp -> number|range2
  (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))))
```

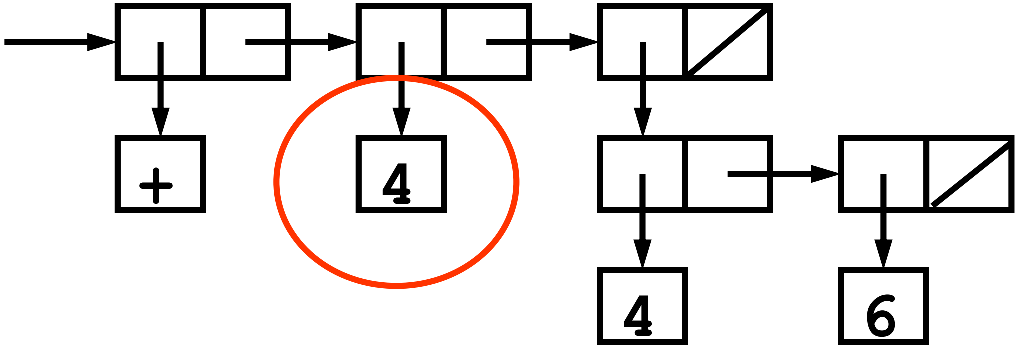
Why is eval-2 broken?

- 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 sums of numbers and ranges (broken!)

```
; Exp = number|range2|SumExp
(define (eval-2 exp) ; type: Exp -> number|range2
  (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))))
```



Range-add-2 expects two ranges, i.e. two lists!

Why is eval-2 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)
```

Key point – doesn't return an error, but gives us what appears to be a legitimate answer!

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? Primitive predicates like **`number?`** and **`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 that 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? (first e) sum-tag)))
```

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

Data abstraction 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? (first e) constant-tag)))

; type: ConstantExp -> number
(define (constant-val const) (second const))
```

3. Eval for numbers with tags (incomplete)

No closure!

```
; Exp = ConstantExp | SumExp
(define (eval-3 exp) ; type: Exp -> number
   (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: Exp -> ConstantExp
(define (eval-4 exp)
 (cond
  ((constant-exp? exp) exp)
  ((sum-exp? exp)
   (make-constant
    (+ (constant-val (eval-4 (sum-addend exp)))
       (constant-val (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

Make add an operation in the Constant abstraction

```
;type: ConstantExp,ConstantExp -> ConstantExp
(define (constant-add c1 c2)
  (make-constant (+ (constant-val c1)
                    (constant-val c2))))

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

Lessons from `eval-3` and `eval-4`

- standard pattern for a data abstration with tagged data
  - a variable stores the tag
  - attach the tag in the constructor
  - write a predicate that checks the tag
    - determines whether an object belongs to the type of the abstraction
  - operations strip the tags, operate, attach the tag again
- must use tagged data everywhere to get full benefits
  - including return values

5. Same pattern: ranges with tags

[3, 7]

```
(define range-tag 'range)

; type: number, number -> RangeExp
(define (make-range min max)
             (list range-tag min max))

; type: anytype -> boolean
(define (range-exp? e)
  (and (pair? e) (eq? (first e) range-tag)))

; type: RangeExp -> number
(define (range-min range) (second range))
(define (range-max range) (third range))
```

5. Eval for numbers and ranges with tags

```
; Exp = ConstantExp | RangeExp | SumExp
(define (eval-5 exp) ; type: Exp -> ConstantExp|RangeExp
  (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))))
```

Simplify eval with a data-directed add function

```
; ValueExp = ConstantExp | RangeExp
(define (value-exp? v)
  (or (constant-exp? v) (range-exp? v)))

; type: ValueExp, ValueExp -> ValueExp
(define (value-add-6 v1 v2)
 (if (and (constant-exp? v1) (constant-exp? v2))
     (constant-add v1 v2)
     (range-add (val2range v1) (val2range v2))))

; val2range: if argument is a range, return it
; else make the range [x x] from a constant x
; This is called coercion
```

Use type coercion to turn constants into ranges

```
(define (val2range val)
   ; type: ValueExp -> RangeExp
   (if (range-exp? val)
        val                    ; just return range
        (make-range (constant-val val)
                    (constant-val val))))
```

6. Simplified eval for numbers and ranges

```
; ValueExp = ConstantExp | RangeExp
; Exp = ValueExp | SumExp
(define (eval-6 exp)
  ; type: Exp -> ValueExp
  (cond
   ((value-exp? exp) exp)
   ((sum-exp? exp)
    (value-add-6 (eval-6 (sum-addend exp))
                 (eval-6 (sum-augend exp))))
   (else (error "unknown expr type: " exp))))
```

Compare eval-6 with eval-1

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

- Compare to eval-1. It is just as simple!

```
(define (eval-1 exp)
  (cond
     ((number? exp)       exp)
     ((sum-exp? exp)
          (+ (eval-1 (sum-addend exp))
             (eval-1 (sum-augend exp))))
     (else
        (error "unknown expression " exp))))
```

- This shows the power of data-directed programming

Eval-7: adding limited-precision numbers

5 +/- 2

```
(define limited-tag 'limited)
(define (make-limited-precision val err)
        (list limited-tag val err))

; Exp = ValueExp | Limited | SumExp
(define (eval-7 exp)
  ; type: Exp -> ValueExp | Limited
  (cond
   ((value-exp? exp) exp)
   ((limited-exp? exp) exp)
   ((sum-exp? exp)
    (value-add-6 (eval-7 (sum-addend exp))
                 (eval-7 (sum-augend exp))))
   (else (error "unknown expr type: " exp))))
```

Oops: `value-add-6` is not defensive

```
(eval-7 (make-sum
          (make-range 4 6)
          (make-limited-precision 10 1)))
 ==> (range 14 16)      WRONG
```

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

- Correct answer should have been **`(range 13 17)`** or **`(limited 15 2)`**

What went wrong in `value-add-6`?

- **`limited-exp`** is not a constant, so falls into the alternative
- **`(limited 10 1)`** passed to **`val2range`**
- **`(limited 10 1)`** passed to **`constant-val`**, returns 10
- **`range-add`** called on **`(range 4 6)`** and **`(range 10 10)`**

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

(define (val2range val)
  (if (range-exp? val)
    val                    ; just return range
    (make-range (constant-val val)  ; assumes constant
                (constant-val val))))
```

7. Defensive version: check tags before operating

```
; type: ValueExp, ValueExp -> ValueExp
(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

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 put code in the else branch of **`if`** or **`cond`**; make it signal an error instead

- Traditionally, operations and accessors don't check tags
  - They assume tags have been checked at the higher level
  - A check in **`constant-val`** would have trapped this bug
  - Be paranoid: add checks in your operations and accessors
    - The cost of redundant checks is usually trivial compared to the cost of your debugging time
  - Andy Grove: "only the paranoid survive"