Structure and Interpretation of Computer Programs

COMP200
• Why do we need tags?
• Concept of tags
• Extended example
The Concept of a Tag

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

\[
\text{(define (make-point x y) (list 'point x y))}
\]
TAGGED DATA

Benefits

- **Data-directed programming:**
  functions that decide what to do based on the arguments
• Data-directed programming:
  functions that decide what to do based on the arguments

  \texttt{area: triangle|square|circle \rightarrow number}
• **Data-directed programming:**
  functions that decide what to do based on the arguments

  \[ \text{area: triangle|square|circle} \rightarrow \text{number} \]

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

Arithmetic Evaluation
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 \texttt{min} and \texttt{max} arithmetic:
Expressions might include values other than numbers

- **Ranges**

  some unknown number between \( \text{min} \) and \( \text{max} \) arithmetic:

  \[
  [3,7] + [1,3] = [4,10]
  \]
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 \( \pm \) some error amount

```scheme
(define exp1 (make-sum (make-sum 3 15) 20))
(exp1) ; ==> (+ (+ 3 15) 20)
(eval-1 exp1) ; ==> 38
```
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 \texttt{min} and \texttt{max} arithmetic:
  
  \[ [3,7] + [1,3] = [4,10] \]

- **Limited precision values**
  some value $\pm$ some error amount
  arithmetic: \((100 \pm 1) + (3 \pm 0.5) = (103 \pm 1.5)\)
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: \((100 \pm 1) + (3 \pm 0.5) = (103 \pm 1.5)\)
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
**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
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-3:7 Do it again with tagged data
1. ABSTRACT DATA TYPE (ADT) for sums

;; type: Exp, Exp -> SumExp
(define (make-sum addend augend) (list '+ addend augend))
1. ABSTRACT DATA TYPE (ADT)
   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) '+)))
1. ABSTRACT DATA TYPE (ADT) 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))
1. ABSTRACT DATA TYPE (ADT) 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

```scheme
;; type: number | SumExp -> number
(define (eval-1 exp)
```
1. **EVAL** FOR NUMBERS ONLY

```scheme
; type: number | SumExp -> number
(define (eval-1 exp)
  (cond
    ((number? exp) exp) ; base case
    ((sum-exp? exp)    ; recursive case
```
1. **EVAL** FOR NUMBERS ONLY

; type: number | SumExp -> number
(define (eval-1 exp)
  (cond
   ((number? exp) exp); base case
   ((sum-exp? exp)       ; recursive case
    (+ (eval-1 (sum-addend exp))
      (eval-1 (sum-augend exp))))))
1. **eval** FOR NUMBERS ONLY

```scheme
; type: number | SumExp -> number
(define (eval-1 exp)
  (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)))))
```
1. **EVAL** FOR NUMBERS ONLY

```lisp
; type: number | SumExp -> number
(define (eval-1 exp)
  (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
```
2. ADT FOR RANGES

(no tags)

; type: number, number -> range2
(define (make-range-2 min max) (list min max))
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))
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
2. **eval** for Numbers and Ranges

(broken)

; type: number|range2|SumExp -> number|range2
(define (eval-2 exp)
  (cond
    ((number? exp) exp))
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))))))
2. **EVAL FOR NUMBERS AND RANGES**

(broken)

```scheme
; 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
```
2. **eval** FOR NUMBERS AND RANGES

(broken)

```scheme
; 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. 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
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!!
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)))
```
3. START AGAIN USING TAGGED DATA

• Take another look at \texttt{SumExp} ... it's already tagged!

\begin{verbatim}
(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)))
\end{verbatim}

• \texttt{sumExp?} is not ambiguous:
  only true for things made by \texttt{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))
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)))
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
3. **EVAL FOR NUMBERS WITH TAGS**

*(incomplete)*

```scheme
; 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))))
```
3. **EVAL FOR NUMBERS WITH TAGS**

*(incomplete)*

```scheme
; 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
```
3. \texttt{EVAL FOR NUMBERS WITH TAGS} \\
\textit{(incomplete)}

\begin{verbatim}
; 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
\end{verbatim}

Not all non-trivial 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
      (+ (constant-val (eval-4 (sum-addend exp)))
       (constant-val (eval-4 (sum-augend exp))))
     ))
    (else (error "unknown expr type: " exp))))
4. **EVAL FOR NUMBERS WITH TAGS**

; type: ConstantExp | SumExp -> ConstantExp
(define (eval-> exp)
  (cond
    ((constant-exp? exp) exp)
    ((sum-exp? exp)
      (make-constant
        (+ (constant-val (eval-> (sum-addend exp)))
          (constant-val (eval-> (sum-augend exp)))))
    (else (error "unknown expr type: " exp))))
4. **EVAL FOR NUMBERS WITH TAGS**

```lisp
; type: ConstantExp | SumExp -> 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))))
```
4. **EVAL FOR NUMBERS WITH TAGS**

```lisp
; type: ConstantExp | SumExp -> 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)
```
4. MAKE **ADD** AN OPERATION IN THE CONSTANT ADT

; 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))))
4. MAKE **ADD** AN OPERATION IN THE CONSTANT ADT

```scheme
; 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))))
```
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 variables
5. SAME PATTERN: RANGE ADT WITH TAGS

(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? (car e) range-tag)))

; type: RangeExp -> number
(define (range-min range) (cadr range))
(define (range-max range) (caddr range))
5. **EVAL** FOR NUMBERS AND RANGES WITH TAGS

; ConstantExp | RangeExp | SumExp
; -> ConstantExp| RangeExp
(define (eval-5 exp)
5. **EVAL FOR NUMBERS AND RANGES WITH TAGS**

; ConstantExp | RangeExp | SumExp
; → ConstantExp| RangeExp
(define (eval-5 exp)
  (cond
    ((constant-exp? exp) exp)
    ((range-exp? exp) exp))
5. **EVAL** FOR NUMBERS AND RANGES WITH TAGS

```scheme
(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)))
   ))
```
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)))))))
```
5. **EVAL FOR NUMBERS AND RANGES WITH TAGS**

; ConstantExp | RangeExp | SumExp
; -> ConstantExp| RangeExp
(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

```
; ValueExp = ConstantExp | RangeExp
(define (value-exp? v)
  (or (constant-exp? v) (range-exp? v)))
```
6. 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
```
6. COERCION TO TURN CONSTANTS INTO RANGES

(define (val2range val)
  (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
; type: ValueExp | SumExp -> ValueExp
(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))))
6. SIMPLIFIED \texttt{EVAL} FOR NUMBERS AND RANGES

\textbf{Compare to eval-1}

; ValueExp = ConstantExp | RangeExp
; type: ValueExp | SumExp \to ValueExp
(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))))

; type: number | SumExp \to number
(define (eval-1 exp)
  (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))))

Power of data-directed programming!
7. **EVAL** FOR ALL DATA TYPES

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

; ValueExp | Limited | SumExp -> ValueExp | Limited
(define (eval-7 exp)
  (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))))
```
7. **VALUE-ADD-6 IS NOT DEFENSIVE**

```lisp
(eval-7 (make-sum
  (make-range 4 6)
  (make-limited-precision 10 1)))

; ==> (range 14 16) WRONG
```
7. VALUE-ADD-6 IS NOT DEFENSIVE

(eval-7 (make-sum
  (make-range 4 6)
  (make-limited-precision 10 1)))

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

  (range 13 17) or (limited 15 2)
7. VALUE-ADD-6 IS NOT DEFENSIVE

• 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: 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
4. LESSONS FROM EVAL-5:7

• Data directed programming can simplify higher level code

• Using tagged data is defensive programming only 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."