6.001: Structure and Interpretation of Computer Programs

- Symbols
- A-lists
- Example of using symbols
- Differentiation

Review: data abstraction

- A data abstraction consists of:
  - constructors
    ```scheme
    (define make-point
      (lambda (x y) (list x y)))
    ```
  - selectors
    ```scheme
    (define x-coor
      (lambda (pt) (car pt)))
    ```
  - operations
    ```scheme
    (define on-y-axis?
      (lambda (pt) (= (x-coor pt) 0)))
    ```
  - contract
    ```scheme
    (x-coor (make-point <x> <y>)) = <x>
    ```

Symbols?

- Say your favorite color
- Say “your favorite color”
- What is the difference?

Creating and Referencing Symbols

- How do I create a symbol?
  ```scheme
  (define alpha 27)
  ```
- How do I reference a symbol’s value?
  ```scheme
  alpha ;Value: 27
  ```
- How do I reference the symbol itself?
  e.g.: How can I build this list: (27 alpha)
  ```scheme
  (list alpha ???) 
  (27 alpha )
  ```

Quote

- Need a way of telling interpreter: “I want the following object as a data structure, not as an expression to be evaluated”
  ```scheme
  (quote alpha) ;Value: alpha
  ```
Symbol: a primitive type
- constructors:
  None since really a primitive not an object with parts
- selectors
  None
- operations:
  symbol? ; type: anytype -> boolean
  (symbol? (quote alpha)) ==> #t
  eq? ; discuss in a minute

A compound proc that squares its argument

The reader converts this expression into an internal representation of the symbol

Symbols are ordinary values
(list 1 2) ==> (1 2)
(list (quote delta) (quote gamma))
  ==> (delta gamma)

A useful property of the quote special form
(list (quote delta) (quote delta))
Two quote expressions with the same name return the same object

The operation eq? tests for the same object
- a primitive procedure
- returns #t if its two arguments are the same object
- very fast

(eq? (quote eps) (quote eps)) ==> #t
(eq? (quote delta) (quote eps)) ==> #f

- For those who are interested:
  eq?: EQtype, EQtype ==> boolean
  EQtype = any type except number or string
- One should therefore use = for equality of numbers, not eq?

Generalization: quoting other expressions
- Expression: Reader converts to: Prints out as:
  1. (quote a)  
  2. (quote (a b))
  3. (quote 1)
  4. (quote "foobar")
Shorthand: the single quote mark

'\(a\) is shorthand for (quote \(a\))

'(1 2) is shorthand for (quote (1 2))

Your turn: what does evaluating these print out?

(define x 20)

(+ x 3) ==>

'(+ x 3) ==> 

(list (quote +) x '3) ==>

(list '+ x 3) ==>

(list + x 3) ==>

23

(+ x 3)

(+ 20 3)

(+ 20 3)

([procedure #…] 20 3)

Davis’ Rule of Thumb for Quote

(list (quote fred (quote quote) (+ 3 5)))

(quote ((quote fred) (quote quote) (+ 3 5))))

What's the value of the quoted expression?

('fred 'quote (+ 3 5))

Traditional LISP structure: association list

• A list where each element is a list of the key and value.

• Represent the table as the alist:

\[
\{(x \ 15) \ (y \ 20)\}
\]
Alist operation: find-assoc

(define (find-assoc key alist)
  (cond
   ((null? alist) #f)
   ((equal? key (caar alist)) (cadar alist))
   (else (find-assoc key (cdr alist)))))

(define a1 '((x 15) (y 20)))
(define (find-assoc 'y a1) ==> 20)

Alist operation: add-assoc

(define (add-assoc key val alist)
  (cons (list key val) alist))

(define a2 (add-assoc 'y 10 a1))
(define (find-assoc 'y a2) ==> 10)

An aside on testing equality

• = tests equality of numbers
• Eq? Tests equality of symbols
  (for now, we’ll see later applies to other structures)
• Equal? Tests equality of symbols, numbers or lists of
  symbols and/or numbers that print the same

Alists are not an abstract data type

• Missing a constructor:
  • Used quote or list to construct
    (define a1 '((x 15) (y 20)))
  • There is no abstraction barrier: the implementation is exposed.
  • User may operate on alists using standard list operations.
  (filter (lambda (a) (< (cadr a) 16)) a1))
    ==> ((x 15))

Why do we care that Alists are not an ADT?

• Modularity is essential for software engineering
  • Build a program by sticking modules together
  • Can change one module without affecting the rest
• Alists have poor modularity
  • Programs may use list ops like filter and map on alists
  • These ops will fail if the implementation of alists change
  • Must change whole program if you want a different table
• To achieve modularity, hide information
  • Hide the fact that the table is implemented as a list
  • Do not allow rest of program to use list operations
  • ADT techniques exist in order to do this

Symbolic differentiation

(deriv <expr> <with-respect-to-var>) ==> <new-expr>

Algebraic expression Representaton
X + 3 (+ x 3)
X X
5y (* 5 y)
X+ y + 3 (+ x (+ y 3))

(deriv (+ x 3) x) ==> 1
(deriv (+ (* x y) 4) x) ==> y
(deriv (+ x x) x) ==> (* x x)
Building a system for differentiation

Example of:
• Lists of lists
• How to use the symbol type
• Symbolic manipulation

1. how to get started
2. a direct implementation
3. a better implementation

1. How to get started
• Analyze the problem precisely
  deriv constant dx = 0
  deriv variable dx = 1 if variable is the same as x
  = 0 otherwise
  deriv (e1+e2) dx = deriv e1 dx + deriv e2 dx
  deriv (e1*e2) dx = e1 * (deriv e2 dx) + e2 * (deriv e1 dx)

• Observe:
  • e1 and e2 might be complex subexpressions
  • derivative of (e1+e2) formed from deriv e1 and deriv e2
  • a tree problem (or list of lists)

Type of the data will guide implementation

• legal expressions
  x
  (+ x y)
  2 (* 2 x)
  (+ (* x y) 3)

• illegal expressions
  * (3 5 +)
  (+ x y z)
  ()
  (3)
  (* x)

: Expr = SimpleExpr | CompoundExpr;
; SimpleExpr = number | symbol;
; CompoundExpr = a list of three elements where the first
  element is either + or *
  = pair< (+|*), pair<Expr, pair<Expr,null> >>

2. A direct implementation
• Overall plan: one branch for each subpart of the type

(define deriv (lambda (expr var)
  (if (simple-expr? expr)
    (if (number? expr) 0
      (if (eq? expr var) 1 0))
    (if (eq? (car expr) '+)
      <handle add expression>
      <handle product expression>)))))

• To implement simple-expr? look at the type
  • CompoundExpr is a pair
  • nothing inside SimpleExpr is a pair
  • therefore
    (define simple-expr? (lambda (e)
      (not (pair? e))))

Simple expressions

• One branch for each subpart of the type

(define deriv (lambda (expr var)
  (if (simple-expr? expr)
    (if (number? expr) 0
      <handle number> 0
      <handle symbol> (if (eq? expr var) 1 0))
    <handle compound expression>)))))

• Implement each branch by looking at the math

Compound expressions

• One branch for each subpart of the type

(define deriv (lambda (expr var)
  (if (simple-expr? expr)
    (if (number? expr) 0
      (if (eq? expr var) 1 0))
    (if (eq? (car expr) '+)
      <handle add expression>
      <handle product expression>)))))
Sum expressions

- To implement the sum branch, look at the math

```Scheme
(define deriv (lambda (expr var)
    (if (simple-expr? expr)
        (if (number? expr) 0
            (if (eq? expr var) 1 0))
        (if (eq? (car expr) '+)
            (list '+
                (deriv (cadr expr) var)
                (deriv (caddr expr) var))
            <handle product expression>))
))
```

- ```Scheme```
  ```Scheme
  (deriv '(+ x y) 'x) ==> (+ 1 0) (a list!)
  ```

The direct implementation works, but...

- Programs always change after initial design
- Hard to read
- Hard to extend safely to new operators or simple exprs
- Can’t change representation of expressions
- Source of the problems:
  - nested if expressions
  - explicit access to and construction of lists
  - few useful names within the function to guide reader

3. A better implementation

1. Use `cond` instead of nested `if` expressions
2. Use data abstraction

- To use `cond`:
  - write a predicate that collects all tests to get to a branch:
    ```Scheme
    (define sum-expr? (lambda (e)
        (and (pair? e) (eq? (car e) '+))))
    ; type: Expr -> boolean
    ```
  - do this for every branch:
    ```Scheme
    (define variable? (lambda (e)
        (and (not (pair? e)) (symbol? e))))
    ```

Use data abstractions

- To eliminate dependence on the representation:

```Scheme
(define make-sum (lambda (e1 e2) (list '+ e1 e2))
(define addend (lambda (sum) (cadr sum)))
```

A better implementation

```Scheme
(define deriv (lambda (expr var)
    (cond
        ([number? expr] 0)
        ([variable? expr] (if (eq? expr var) 1 0))
        ([sum-expr? expr]
            (make-sum (deriv (addend expr) var)
                      (deriv (augend expr) var))
        ([product-expr? expr]
            <handle product expression>)
        (else
            (error "unknown expression type" expr)))
    )
))
```

Dispatch on type

- ```Scheme```
  ```Scheme
  (deriv '(+ x y) 'x) ==> (+ 1 0) (a list!)
  ```

Isolating changes to improve performance

- ```Scheme```
  ```Scheme
  (deriv '(+ x y) 'x) ==> (+ 1 0) (a list!)
  ```
  ```Scheme
  (define make-sum (lambda (e1 e2) (list '+ e1 e2))
  (define addend (lambda (sum) (cadr sum)))
  ```

- ```Scheme```
  ```Scheme
  (deriv '(+ x y) 'x) ==> 1
  ```
Modularity makes changes easier

- So it seems like a bit of a pain to be using expressions like: (+ 2 x) or (* (+ 3 x) (+ x y))
- It would be cleaner somehow to use more algebraic expressions, like: (2 + x) or ((3 + x) * (x + y))
- What do we need to change?

Just change data abstraction

- Constructors
  ```scheme```
  ```
  (define (make-sum e1 e2)
    (list e1 '+ e2))
  ```
  ```
  (define (augend expr)
    (car expr))
  ```
  ```
  (define (sum-expr? Expr)
    (and (pair? Expr) (eq? '+ (cadr expr))))
  ```
  ```
  ```
```

- Accessors
  ```scheme```
  ```
  (define (augend expr)
    (car expr))
  ```
  ```
  (define (sum-expr? Expr)
    (and (pair? Expr) (eq? '+ (cadr expr))))
  ```

- Predicates
  ```scheme```
  ```
  ```
  ```

Modularity helps in other ways

- Rather than changing the code to handle simplifications of expressions, write a separate simplifier:
  ```scheme```
  ```
  (define (simplify expr)
    (cond ((sum-expr? expr)
            (simplify-sum expr))
          ((product-expr? expr)
            (simplify-product expr))
          (else expr)))
  ```
  ```
  ```
  ```
```

Separating out aspects of simplification

- Rather than changing the code to handle simplifications of expressions, write a separate simplifier:
  ```scheme```
  ```
  ```
  ```
  ```
```

```scheme```
  ```
  ```
  ```
  ```
```
  ```
  ```
  ```
  ```
```

```scheme```
  ```
  ```
  ```
  ```
```

```scheme```
  ```
  ```
  ```
  ```
```

```scheme```
  ```
  ```
  ```
  ```
```

```scheme```
  ```
  ```
  ```
  ```
```

```scheme```
  ```
  ```
  ```
  ```
```

```scheme```
  ```
  ```
  ```
  ```
```

```scheme```
  ```
  ```
  ```
  ```
```

```scheme```
  ```
  ```
  ```
  ```
```

```scheme```
  ```
  ```
  ```
  ```
```

```scheme```
  ```
  ```
  ```
  ```
```

```scheme```
  ```
  ```
  ```
  ```
```

```scheme```
  ```
  ```
  ```
  ```
```

```scheme```
  ```
  ```
  ```
  ```
```

```scheme```
  ```
  ```
  ```
  ```
```

```scheme```
  ```
  ```
  ```
  ```
```

```scheme```
  ```
  ```
  ```
  ```
```

```scheme```
  ```
  ```
  ```
  ```
```

```scheme```
  ```
  ```
  ```
  ```
```

```scheme```
  ```
  ```
  ```
  ```
```

```scheme```
  ```
  ```
  ```
  ```
```

```scheme```
  ```
  ```
  ```
  ```
```

```scheme```
  ```
  ```
  ```
  ```
```

```scheme```
  ```
  ```
  ```
  ```
```

```scheme```
  ```
  ```
  ```
  ```
```

```scheme```
  ```
  ```
  ```
  ```
```

```scheme```
  ```
  ```
  ```
  ```
```

```scheme```
  ```
  ```
  ```
  ```
```

```scheme```
  ```
  ```
  ```
  ```
```

```scheme```
  ```
  ```
  ```
  ```
```

```scheme```
  ```
  ```
  ```
  ```
```

```scheme```
  ```
  ```
  ```
  ```
```

```scheme```
  ```
  ```
  ```
  ```
```

```scheme```
  ```
  ```
  ```
  ```
```

```scheme```
  ```
  ```
  ```
  ```
```

```scheme```
  ```
  ```
  ```
  ```
```

```scheme```
  ```
  ```
  ```
  ```
```

```scheme```
  ```
  ```
  ```
  ```
```

```scheme```
  ```
  ```
  ```
  ```
```

```scheme```
  ```
  ```
  ```
  ```
```

```scheme```
  ```
  ```
  ```
  ```
```

```scheme```
  ```
  ```
  ```
  ```
```

```scheme```
  ```
  ```
  ```
  ```
```

```scheme```
  ```
  ```
  ```
  ```
```

```scheme```
  ```
  ```
  ```
  ```
```

```scheme```
  ```
  ```
  ```
  ```
```

```scheme```
  ```
  ```
  ```
  ```
```

```scheme```
  ```
  ```
  ```
  ```
```

```scheme```
  ```
  ```
  ```
  ```
```

```scheme```
  ```
  ```
  ```
  ```
```

```scheme```
  ```
  ```
  ```
  ```
```

```scheme```
  ```
  ```
  ```
  ```
```

```scheme```
  ```
  ```
  ```
  ```
```