Knowledge-based systems in Bioinformatics, 1MB602, 2006

Lecture 4:
Symbolic and abstract data,
assignments and local state

Lecture overview

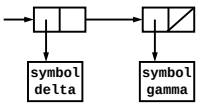
- Quotation
- Symbolic differentiation
- Complex numbers
- Tagged data
- Data directed programming
- Message passing
- Imperative programming
- Local state

Symbols?

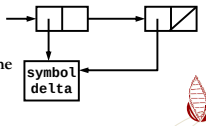
- Say your favorite color
- Say "your favorite color"
- What is the difference?
  - In one case, we want the meaning associated with the expression
  - In the other case, we want the actual words (or symbols) of the expression
- Constructors:
  - `(quote alpha)` → "alpha"
  - `quote` is a special form. One argument: a name.
- Operations:
  `(symbol? (quote alpha))` → `#t`

Symbols are ordinary values

```
(list 1 2)      →  (1 2)
(list (quote delta) (quote gamma))
   → (delta gamma)
```




```
(list (quote delta) (quote delta))
   → (delta delta)
```



two quote expressions with the same name return the same object

The eq? operation

- Primitive procedure
- Returns `#t` if the two arguments are the same object

```
(eq? (quote eps) (quote eps))   → #t
(eq? (quote delta) (quote eps)) → #f
```


Quoting

- Shorthand
  - `'a` is shorthand for `(quote a)`
  - `'(1 2)` is shorthand for `(quote (1 2))`
- Rewritings
  - `(quote (a b))` → `(list (quote a) (quote b))`
  - `(quote <self-eval-expr>)` → `<self-eval-expr>`

```
- (quote (1 (b c)))   → (list (quote 1)
                             (quote (b c)))
                      → (list 1
                             (list (quote b)
                                   (quote c)))
                      → (1 (b c))
```

Quoting cont.

```
(define x 20)

   (+ x 3)                  → 23

   '(+ x 3)                 → (+ x 3)

   (list (quote +) x '3)    → (+ 20 3)

   (list '+ x 3)            → (+ 20 3)

   (list + x 3)             → ([procedure #…] 20 3)
```


Symbolic differentiation

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

   (deriv '(+ x 3) 'x)       → 1
   (deriv '(+ (* x y) 4) 'x) → y
   (deriv '(* x x) 'x)       → (+ x x)
```

- How to implement?

Symbolic differentiation cont.

- How to get started?
- Analyze the problem precisely

base:
- deriv constant dx = 0
- deriv variable dx = 1 if variable is the same as x
  = 0 otherwise

recursive:
- 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
  - Derivate of (e1+e2) is formed from deriv e1 and deriv e2
  - A tree problem

Symbolic differentiation cont.

- Wishful thinking
  - Assume we have means for representing algebraic expressions
  - Assume we have procedures to implement the following

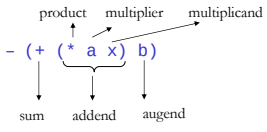
`(variable? e)`	Is e a variable?
`(same-variable? v1 v2)`	Are v1 and v2 the same variable?
`(sum? e)`	Is e a sum?
`(addend e)`	Addend of the sum e
`(augend e)`	Augend of the sum e
`(make-sum a1 a2)`	Construct the sum of a1 and a2
`(product? e)`	Is e a product?
`(multiplier e)`	Multiplier of the product e
`(multiplicand e)`	Multiplicand of the product e
`(make-product m1 m2)`	Construct the product of m1 and m2

The differentiation procedure

```
(define (deriv exp var)
  (cond ((number? exp) 0)                                   } base
        ((variable? exp)
         (if (same-variable? exp var) 1 0))
        ((sum? exp)                                         } recursive
         (make-sum (deriv (addend exp) var)
                   (deriv (augend exp) var)))
        ((product? exp)
         (make-sum
            (make-product (multiplier exp)
                          (deriv (multiplicand exp) var))
            (make-product (deriv (multiplier exp) var)
                          (multiplicand exp))))
        (else
         (error "unknown expr type -- DERIV" exp))))
```

The differentiation procedure cont.

- The `deriv` procedure incorporates the complete differentiation algorithm
- Expressed in terms of abstract data
  - It will work no matter how we choose to represent algebraic expressions!
- How to represent the algebraic expressions?
- How to represent $ax + b$?
  - `(a * x + b)`



Representing algebraic expressions

- Variables are symbols

```
(define (variable? x) (symbol? x))
(define (same-variable? v1 v2)
    (and (variable? v1) (variable? v2) (eq? v1 v2)))
```

- Sums and products are constructed as lists

```
(define (make-sum a1 a2) (list '+ a1 a2))
(define (make-product m1 m2) (list '* m1 m2))
```

- Sums and products are identified by their first element

```
(define (sum? x)
    (and (pair? x) (eq? (car x) '+)))
(define (product? x)
    (and (pair? x) (eq? (car x) '*)))
```


Representing algebraic expressions cont.

- Addends and multipliers are the second items of the sum and the product lists

```
(define (addend s) (car (cdr s)))
(define (multiplier p) (car (cdr p)))
```

- Augends and multiplicands are the third items of the sum and the product lists

```
(define (augend s) (car (cdr (cdr s))))
(define (multiplicand p) (car (cdr (cdr p))))
```

- Done!

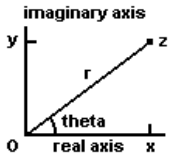
Complex numbers

- Task: develop a system that performs arithmetic operations on complex numbers
- Two plausible representations of complex numbers
  - Rectangular form: (real part and imaginary part)
    $z = x + iy$
  - Polar form (magnitude and angle)
    $z = r\, e^{i*\theta}$
- How did we represent rational numbers?
- Data abstraction!
  - Operations on complex numbers should work regardless of the representation!
  - Wishful thinking, assume it is already done

Complex numbers cont.

- Assume that the operations on complex numbers are implemented in terms of four selectors:

```
real-part
imag-part
magnitude
angle
```



- Addition (rectangular form)
  Real-part(z1 + z2) = Real-part(z1) + Real-part(z2)
  Imag-part(z1 + z2) = Imag-part(z1) + Imag-part(z2)
- Multiplication (polar form)
  Magnitude(z1 * z2) = Magnitude(z1) * Magnitude(z2)
  Angle(z1 * z2) = Angle(z1) + Angle(z2)

Arithmetic on complex numbers

- Assume we have two constructors of complex numbers
  `make-from-real-imag` (from rectangular parts)
  `make-from-mag-ang` (from polar parts)
- Implement addition and subtraction in terms of real and imaginary parts from rectangular form

```
(define (add-complex z1 z2)
    (make-from-real-imag
          (+ (real-part z1) (real-part z2))
          (+ (imag-part z1) (imag-part z2))))

(define (sub-complex z1 z2)
    (make-from-real-imag
          (- (real-part z1) (real-part z2))
          (- (imag-part z1) (imag-part z2))))
```


Arithmetic on complex numbers cont.

- Implement multiplication and division in terms of magnitude and angle parts from polar form

```
(define (mul-complex z1 z2)
    (make-from-mag-ang
          (* (magnitude z1) (magnitude z2))
          (+ (angle z1) (angle z2))))

(define (div-complex z1 z2)
    (make-from-mag-ang
          (/ (magnitude z1) (magnitude z2))
          (- (angle z1) (angle z2))))
```

- Done?
- Done with wishful thinking

Representing complex numbers

- How to represent complex numbers?
  - Rectangular form?
  - Polar form?
- What if we want both representations?
  - We need a way to distinguish the type of the complex number
  - Attach a type tag on the complex number
- 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 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!

Tagging data

- Attach tags on data

```
(define (attach-tag type-tag contents)
    (cons type-tag contents))
```

- Get the tag

```
(define (type-tag datum)
    (if (pair? datum) (car datum)
        (error "Bad tagged datum" datum)))
```

- Get the data

```
(define (contents datum)
    (if (pair? datum) (cdr datum)
        (error "Bad tagged datum" datum)))
```


Tagging complex numbers

- Checking complex types

```
(define (rectangular? z)
    (eq? (type-tag z) 'rectangular))
(define (polar? z)
    (eq? (type-tag z) 'polar))
```

- Constructors
  - Construct rectangular form from real and imaginary
  - Construct rectangular form from magnitude and angle
  - Construct polar form from real and imaginary
  - Construct polar form from magnitude and angle

$$x = r * \cos(\phi) \quad r = \sqrt{x^2 + y^2}$$
$$y = r * \sin(\phi) \quad \phi = \arctan(y, x)$$

Constructing complex numbers

- Constructing rectangular forms

```
(define (make-rect-from-real-imag x y)
    (attach-tag 'rectangular (cons x y)))
(define (make-rect-from-mag-ang r t)
    (attach-tag 'rectangular
          (cons (* r (cos t))
                (* r (sin t)))))

(define (make-polar-from-mag-ang r t)
    (attach-tag 'polar (cons r t)))
(define (make-polar-from-real-imag x y)
    (attach-tag 'polar
          (cons (sqrt (+ (square x) (square y)))
                (atan y x))))
```


Implementing selectors

- Real part

```
(define (real-part z)
    (cond ((rectangular? z)
           (real-part-rect (contents z)))
          ((polar? z)
           (real-part-polar (contents z)))
          (else (error "Unknown type: REAL-PART" z))))

(define (real-part-rect z) (car z))

(define (real-part-polar z)
    (* (magnitude-polar z) (cos (angle-polar z))))
```

Implementing selectors cont.

- Imag part

```
(define (imag-part z)
    (cond ((rectangular? z)
           (imag-part-rect (contents z)))
          ((polar? z)
           (imag-part-polar (contents z)))
          (else (error "Unknown type: IMAG-PART" z))))

(define (imag-part-rect z) (cdr z))

(define (imag-part-polar z)
    (* (magnitude-polar z) (sin (angle-polar z))))
```


Implementing selectors cont.

- Magnitude

```
(define (magnitude z)
    (cond ((rectangular? z)
           (magnitude-rect (contents z)))
          ((polar? z)
           (magnitude-polar (contents z)))
          (else (error "Unknown type: MAGNITUDE" z))))

(define (magnitude-rect z)
    (sqrt (+ (square (real-part-rect z))
             (square (imag-part-rect z)))))

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


Implementing selectors cont.

- Angle

```
(define (angle z)
    (cond ((rectangular? z)
           (angle-rect (contents z)))
          ((polar? z)
           (angle-polar (contents z)))
          (else (error "Unknown type: ANGLE" z))))

(define (angle-rect z)
    (atan (imag-part-rect z)
          (real-part-rect z)))

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


Constructing complex numbers

- Construct from real and imaginary → rectangular form

```
(define (make-from-real-imag x y)
    (make-rect-from-real-imag x y))
```

- Construct from magnitude and angle → polar form

```
(define (make-from-mag-ang r t)
    (make-polar-from-mag-ang r t))
```

- Our operations still work!

```
(define (add-complex z1 z2)
    (make-from-real-imag
          (+ (real-part z1) (real-part z2))
          (+ (imag-part z1) (imag-part z2))))
```


Data directed programming

Types

Operations	Polar	Rectangular
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

Message passing

Instead of "intelligent operations" that dispatch on data types, what about "intelligent data objects" that dispatch on operation names?

```
(define (make-from-real-imag x y)
  (define (dispatch op)
      (cond ((eq? op 'real-part) x)
            ((eq? op 'imag-part) y)
            ((eq? op 'magnitude)
             (sqrt (+ (square x) (square y))))
            ((eq? op 'angle) (atan y x))
            (else
             (error "Unknown op" op))))
  dispatch)
```

Factorial procedure

- Functional design

```
(define (fac n)
   (define (iter count prod)
      (if (> count n) prod
          (iter (+ count 1) (*count prod))))
   (iter 1 1))
```

- Imperative design

```
(define (fac n)
   (let ((count 1) (prod 1))
      (define (loop)
         (if (> count n) prod
             (begin
               (set! prod (* count prod))
               (set! count (+ count 1))
               (loop))))
      (loop)))
```


Assignments: `set!`

- Substitution model -- *functional programming*:

```
(define x 10)
(+ x 5) → 15
...
(+ x 5) → 15
```

- expression has the same value each time it is evaluated

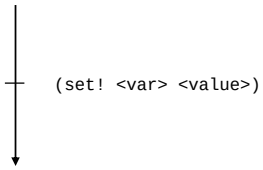
- With assignment:

```
(define x 10)
(+ x 5) → 15
...
(set! x 94)
...
(+ x 5) → 99
```

- expression's "value" depends on when it is evaluated

Assignments

- Assignments introduces time in our models



- When an assignment occurs the state of the modeled system changes

Decrement – no assignment

```
(define (make-decrementer balance)
    (lambda (amount)
          (- balance amount)))

(define D (make-decrementer 25))

(D 20)   → 5
(D 10)   → 15
```

- Substitution model:

```
((make-decrementer 25) 20)
((lambda (amount) (- 25 amount)) 20)
(- 25 20)
5
```


Withdraw – with assignment

```
(define (make-simple-withdraw balance)
   (lambda (amount)
      (set! balance (- balance amount))
      balance))
(define W (make-simple-withdraw 25))

(W 20)   → 5
(W 10)   → -5
```

- Substitution model:

```
((make-simple-withdraw 25) 20)
((lambda (amount) (set! balance (- 25 amount)) 25) 20)
((set! balance (- 25 20)) 25)
((set! balance 5) 25)
```

- What???

Substitution model and `set!`

- Why doesn't the substitution model work with assignments?
  - The substitution model is based on the assumption that the symbols in the language are names that refer to values
  - If `set!` is included then the symbols will no longer be names that refer to values but names that refer to locations where values are stored
  - The value of such a location can be changed by `set!`
  - To cope with this new situation we introduce the environment model of evaluation

Environment model of evaluation

- What is it?
- A precise, completely mechanical description of:
  - Name-rule: looking up the value of a variable
  - Define-rule: creating a new definition of a variable
  - Set!-rule: changing the value of a variable
  - Lambda-rule: creating a procedure
  - Application: applying a procedure
- New viewpoint
  - Variable
    - Location where a value is stored
  - Procedure
    - Object with inherited context
  - Expression
    - Only have meaning with respect to an environment

Message passing

Simulating cons-cells with message passing to objects

```
(define (cons car-part cdr-part)
  (define (set-car-part! new-val)
     (set! car-part new-val))
  (define (set-cdr-part! new-val)
     (set! cdr-part new-val))
  (define dispatch (mess)
     (cond ((eq? mess 'car) car-part)
           ((eq? mess 'cdr) cdr-part)
           ((eq? mess 'set-car!)
            set-car-part!)
           ((eq? mess 'set-cdr!)
            set-cdr-part!)
           (else "bad message")))
  dispach)
```


Message passing cont.

```
(define (car cons-pair)
  (cons-pair 'car))

(define (cdr cons-pair)
  (cons-pair 'cdr))

(define (set-car! cons-pair new-val)
  ((cons-pair 'set-car!) new-val))

(define (set-cdr! cons-pair new-val)
  ((cons-pair 'set-cdr) new-val))
```


References

- H. Abelson, G.J. Sussman, Structure and Interpretation of Computer Programs 2nd ed, The MIT Press, Cambridge, Massachusetts, 2000, Chp: 2.3.1-2.3.3, 2.4, 3.0-3.2 pp 142-161, 169-187, 217-251
- 6.001 Spring 2000: Lecture Notes, lecture 7,8,11 http://sicp.ai.mit.edu/Spring-2000/lectures/