Principles of Programming Languages

CS510

Scheme

Types and Type Annotation

Pattern Matching

Higher-Order Functions

Scheme

Scheme is a functional language:

- Programs are expressions
- Expressions (formal) denote values (abstract)
- The value denoted by an expression is obtained by evaluating it
- Expressions may be atomic or composite
  - We'll see examples in a minute
- Expressions are typed, but not statically

Atomic Expressions

- Atomic expressions evaluate to themselves
  - They are also coined values
- The most important ones are:
  - Numbers: 2
  - Booleans: `#t`, `#f`
  - Chars: `#\A`, `#\B`
  - Strings: `"Hello"`
  - Symbols: 'Hello
  - Functions[1]: `(lambda (x) (+ x 1))`
  - Also: Pairs (of values), Lists (of values), Vectors (of values)

[1] It actually evaluates to `#<procedure>`

Symbols

- Symbols: an atomic value that prints like an identifier
- It must be preceded with a quote and start with an identifier

```
> 'a
'a

> 'hello
'hello
```

- `'1234` is not a symbol, it is a number

Pairs

```
> '(1 . 2)
'(1 . 2)

> '(#t . 3)
'(#t . 3)

> '( (1 . 2) . #t)
'((1 . 2) . #t)
```

Lists

```
> '(1 2 3 4)
'(1 2 3 4)

> '(#t 2 3 4)
'(#t 2 3 4)

> '()
'()

> '((2 3) #t)
'((2 3) #t)
```

Functions

- Functions are called procedures in Scheme
- We'll use both names interchangeably

```
> (lambda (x) (+ x 1))
#<procedure>

> (lambda (x) (+ x x))
#<procedure>

> (lambda (x y) (+ x y))
#<procedure>
```

- The leftmost occurrence of `x` in `(lambda (x) (+ x 1))` is called a formal parameter (more on this later)

Vectors

```
> #(1 2 3)
'#(1 2 3)

> #(1 #t '(1 2))
'#(1 #t '(1 2))
```

- Constant time access
- Fixed length

Composite Expressions

- Having finished with our overview of atomic expressions, we now shift to composite expressions
- These are constructed out of other, simpler expressions

```
> (+ 2 3)
5
> (> 2 3)
#f

> (integer? 2) ;; type-checking integers
#t
```

Composite Expressions

- Some more examples of composite expressions

```
> (boolean? 2)
#f

> (boolean? #t)
#t

> (boolean? (+ 2 3))
#f

> (eq? 2 3)
#f
```

- More examples follow; they are grouped according to the type of the expressions thay manipulate

Operations on Pairs

```
> (pair? '(2 . 3)) ;; observer (type-checking pairs)
#t

> (cons 1 2)  ;; constructor
'(1 . 2)

> (car (cons 1 2)) ;; destructor
1

> (cdr (cons 1 2)) ;; destructor
2
```

Operations on Lists

```
> (list? '(1 . 2))  ;; observer
#f
> (list? '(1 2))
#t
> (null? '())
#t
> (null? '(1))
#f

> null
'()
> (cons 1 (cons 2 null))
'(1 2)
> (car (cons 1 (cons 2 null)))
1
> (cdr (cons 1 (cons 2 null)))
'(2)
```

Operations on Symbols

```
> (eq? 'a 'a)
#t

> (eq? 'a 'A)
#f

> (symbol? '(1 2 3))
#f

> (symbol? '1)
#f

> (symbol? 'hello)
#t
```

Operations on Vectors

```
> (vector 1 2)
'#(1 2)

> (make-vector 5)
'#(0 0 0 0 0)

> (vector? 2)
#f

> (vector? #(1 2 3))
#t

> (vector-ref #(5 67 98) 2)
98

> (vector-length #(5 67 98))
3
```

Operations on Functions

- Functions are applied to arguments
- Application is given by juxtaposition and surrounding parenthesis
- We have already seen some built-in functions being applied, eg. `(car '(1 2 3))`
- We can also apply our own functions

```
> ((lambda (x y) (+ x y)) 3 4)
7

> ((lambda (x) (+ x 1)) 3)
4
```

Higher-Order Functions

- These are functions that either receive functions as arguments or return functions as results

```
> ((lambda (x) (x 1)) (lambda (y) (+ y 1)))
2
```

- This feature adds significant expressive power to Scheme
- Later we shall see some important examples of higher-order functions

Evaluation

- Consider an expression such as

  `(e1 e2)`

- Its denoted value is obtained by:

  1. Evaluating `e1` to a value `v1` (which should be a function, otherwise an error is generated)
  2. Evaluating `e2` to a value `v2`
  3. Replacing the formal parameter of `v1` with `v2`

- The fact that `e2` is evaluated before passing its value on to `v1` means that Scheme uses a call-by-value evaluation strategy

```
> ((lambda (x y) (x y)) (lambda (z) (+ z 1)) 2)
3
```

Naming Expressions – Local Definitions

```
> (define x 3)
> x
3
> (+ x 1)
4

> (define succ (lambda (x) (+ x 1)))
> (succ 3)
4

> (define add (lambda (x y) (+ x y)))
> (add 1 3)
4
> add
#<procedure:add>
```

Conditional – if-then-else

```
> (if (> 2 3) 3 5)
5

> (define not (lambda (x) (if x #f #t)))
> (not #t)
#f
> (not #f)
#t
```

Conditional – cond

```
> (cond)
> (cond
    [else 5])
5
> (cond
   [(positive? -5) (error "doesn't get here")]
   [(zero? -5) (error "doesn't get here, either")]
   [(positive? 5) 'here])
'here
> (cond
   [(member 2 '(1 2 3)) => (cdr l)])
'(3)
> (cond
   [(member 2 '(1 2 3))])
'(2 3)
```

Typed Expressions

- Expressions in Scheme are typed
- To determine whether an expression has a specific type we can use type-checking predicates
- A predicate is a function that returns a boolean

Numbers	`number?`
Booleans	`boolean?`
Chars	`char?`
Strings	`string?`
Symbols	`symbol?`
Pairs	`pair?`
Lists	`list?`
Vectors	`vector?`
Function	`procedure?`

Predicates for Testing the Type of an Expression

- Here are some examples we've already seen

```
> (boolean? 2)
#f

> (boolean? #t)
#t

> (boolean? (+ 2 3))
#f

> (symbol? '(1 2 3))
#f

> (symbol? 'hello)
#t
```

Documenting Function Types

- It is convenient to document the type of a function
- The type of a function is a synthesis of its behavior
- This may be done as a comment, using Scheme's own type predicates

```scheme
;; procedure? number? number?
(define f (lambda (x) (+ x x)))

;; procedure? (pair? boolean? boolean?) boolean?
(define and2 (lambda (x y) (and x y)))
```

Documenting Function Types

- However, we will use an alternative, abbreviated notation
- Rather than this:

```
;; procedure? number? number?
(define f (lambda (x) (+ x x)))

;; procedure? (pair? boolean? boolean?) boolean?
(define and2 (lambda (x y) (and x y)))
```

- We write

```
;;  num -> num
(define f (lambda (x) (+ x x)))

;; (bool,bool) ->  bool
(define and2 (lambda (x y) (and x y)))
```

Typing functions with multiple arguments

```
;; {num,num} -> num
(define (add x y)
     (+ x y))
```

Compare with

```
;; num -> (num -> num)
(define add2
   (lambda (x)
      (lambda (y)
         (+ x y))))
```

Type Abbreviations

- Rather than `number?` we write `num`
- Rather than `boolean?` we write `bool`
- Rather than `char?` we write `char`
- Rather than write `procedure? num bool` we shall write `num -> bool`
- Rather than write `pair? num bool` we shall write `(num,bool)`
- Rather than write `list? num` we shall write `[num]`
- Another Example:

  ```
  [(num,bool)] -> char
  ```

  is the type of a function that given a list of pairs of numbers and booleans returns a char.

Type Variables

- We use type variables to denote an arbitrary type
- For example,

```
;; (a,b) -> (b,a)
;; [a] -> num
```

- The first is the type of `swap`, a function that swaps the components of a pair
- The second could be type of length

Pattern Matching – On Lists

- Pattern matching gives access to the components of a value

```
> (define pmL
    (lambda (xs)
      (match xs
        ['() "empty list"]
        [(cons h ys) "non-empty list"]
        [_ "not a list"]
      )))

> (pmL 3)
"not a list"
> (pmL '(1))
"non-empty list"
> (pmL 3)
"not a list"
```

Pattern Matching – On Lists

- Pattern matching gives access to the components of a value

```
> (define pmL
    (lambda (xs)
      (match xs
        ['() "empty list"]
        [(cons h ys) "non-empty list"]
        [_ "not a list"]
      )))

> (pmL 3)
"not a list"
> (pmL '(1))
"non-empty list"
> (pmL 3)
"not a list"
```

Pattern Matching – On Lists

- Pattern matching gives access to the components of a value

```
> (define pmL
    (lambda (xs)
      (match xs
        ['() "empty list"]
        [(cons h ys) "non-empty list"]
        [_ "not a list"]
      )))

> (pmL 3)
"not a list"
> (pmL '(1))
"non-empty list"
> (pmL 3)
"not a list"
```

Pattern Matching – On Lists

- Pattern matching gives access to the components of a value

```
> (define pmL
    (lambda (xs)
      (match xs
        ['() "empty list"]
        [(cons h ys) "non-empty list"]
        [_ "not a list"]
      )))

> (pmL 3)
"not a list"
> (pmL '(1))
"non-empty list"
> (pmL 3)
"not a list"
```

Pattern Matching – On Lists

- Pattern matching gives access to the components of a value

```
;; [a] -> bool
(define pmL
  (lambda (xs)
    (match xs
      ['() "empty list"]
      [(cons h ys) "non-empty list"]
   ;; [_ "not a list"] Dropped due to type annotation
    )))
```

Pattern Matching – On Lists

- Pattern matching gives access to the components of a value

```
;; [a] -> bool
(define pmL
  (lambda (xs)
    (match xs
      ['() "empty list"]
      [(cons h ys) "non-empty list"]
   ;; [_ "not a list"] Dropped due to type annotation
    )))
```

Pattern Matching – On Lists

```
> (define pmL2
  (lambda (xs)
    (match xs
      ['() "empty list"]
      [(cons h ys) h]
      [_ "not a list"]
      )))

> (pmL2 '(1 2 3))
1
```

Pattern Matching – On Pairs

```
> (define pmP
   (lambda (xs)
    (match xs
     [(cons l r) "a pair"]
      [_ "not a pair"]
      )))

> (pmP 1)
"not a pair"
> (pmP '(1))
"a pair"
> (pmP '(1 2 3))
"a pair"
> (pmP '(1 . 2))
"a pair"
```

- Note that lists are nested pairs ending in an empty list
- `'(1 2)` is the same as `'( 1 . (2 . ()))`

An example with numbers

```
;; pos -> pos
> (define fact
    (lambda (n)
      (match n
        [0 1]
        [m (* m (fact (- m 1)))]
      )))

> (fact 5)
120
> (fact 0)
1
> (fact "hello")
-: contract violation
```

Scheme

Types and Type Annotation

Pattern Matching

Higher-Order Functions

More Examples of Recursion in Scheme – map, filter, foldr

- We take a look at some basic higher-order functions:
  - map
  - filter
  - foldr
- We motivate them through concrete examples
- Then we synthesize definitions for them

Motivating `map`

Consider how we would define the following functions:

- `upperL`: Convert each char in a list to uppercase. Eg.

```
> (upperL '( #\a #\B))
(#\A #\B)
```

- `zeroL`: For each number in a list, test it for zero. Eg.

```
> (zeroL '(1 2 0 3 4 ))
'(#f #f #t #f #f)
```

- `succL`: Add one to each number in a list of numbers. Eg.

```
> (succL '(1 2 3))
(2 3 4)
```

Definitions using Structural Recursion

```
;; upperl :: [char] -> [char]
(define upperL
  (lambda (xs)
    (match xs
        ['() '()]
        [(cons h ys) (cons (char-upcase h) (upperL
    ys))]
      )))
;; zeroL :: [num] -> [bool]
(define zeroL
  (lambda (xs)
    (match xs
        ['() '()]
        [(cons h ys) (cons (zero? h) (zeroL ys))]
      )))
;; succL :: [num] -> [num]
(define succL
  (lambda (xs)
    (match xs
        ['() '()]
        [(cons h ys) (cons (succ h) (succL ys))]
      )))
```

Defining `map`

```
(define map
  (lambda (f xs)
    (match xs
      ['() '()]
      [(cons h ys) (cons (f h) (map f ys))]
      )))
```

- `map` is higher-order
  - Its parameter `f` is a function
- What is the type of `map`?

Defining `map`

```
(define map
  (lambda (f xs)
    (match xs
      ['() '()]
      [(cons h ys) (cons (f h) (map f ys))]
      )))
```

- `map` is higher-order
  - Its parameter `f` is a function
- What is the type of `map`? `;; (a->b) -> [a] -> [b]`

Rewriting the examples using `map`

```
;; upperl :: [char] -> [char]
(define (upperL xs)
    (map char-upcase xs)
  )

;; zeroL :: [num] -> [bool]
(define (zeroL xs)
    (map zero? xs)
  )

;; succL :: [num] -> [num]
(define (succL xs)
    (map succ xs)
  )
```

Motivating `filter`

- `filterPositive`: Drop all non-positive numbers from a list. Eg.

```
> (filter positive? '(1 -3 4))
'(1 4)
```

- `filterNonNull`: Drop all non-null lists from a list of lists. Eg.

```
> (filter (compose not null?) '((1 2) (#t) ()
    (#\a #\& #\2)))
'((1 2) (#t) (#\a #\& #\2))
```

Defining examples using structural recursion

```
;; [num] -> [num]
(define filterPositive
  (lambda (xs)
    (match xs
      ['() '()]
      [(cons h ys) (if (positive? h)
                       (cons h (filterPositive ys))
                       (filterPositive ys))]
      )))
;; [[a]] -> [[a]]
(define filterNonNull
  (lambda (xs)
    (match xs
      ['() '()]
      [(cons h ys) (if (not (null? h))
                       (cons h (filterNonNull ys))
                       (filterNonNull ys))]
      )))
```

- `a` in the type for `filterNonNull` stands for any type
- What do you these have in common?

Useful higher-order functions: filter

```
(define filterPositive
  (lambda (xs)
    (filter positive? xs)
   ))

(define filterNonNull
  (lambda (xs)
    (filter (compose not null?) xs)
  ))
```

A shorter version is:

```
(define filterPositive ((curry filter) positive?))

(define filterNonNull ((curry filter) (compose not
    null?)))
```

- Read about `curry`

Motivating `foldr`

- `sumL`: Adds the numbers in a list. Eg.

```
;; [num] -> num
> (sumL '(1 2 3))
6
```

- `allEven`: Determines whether all numbers in a list are even

```
;; [num] -> bool
> (allEven '(1 2 3))
#f

> (allEven '(2 4 6))
#t
```

- `concat`: concatenates all the lists in a list of lists.

```
;; [[a]] -> [a]
> (concat '((1 2) '() '((4))))
'(1 2 (4))
```

Definitions using Structural Induction

```
;; [num] -> num
(define sumL
  (lambda (xs)
    (match xs
        ['() 0]
        [(cons h ys) (+ h (sumL ys))]
      )))
;; [num] -> bool
(define allEven
  (lambda (xs)
    (match xs
        ['() #t]
        [(cons h ys) (and (even? h) (allEven ys))]
      )))
;; [[a]] -> [a]
(define concat
  (lambda (xs)
    (match xs
        ['() '()]
        [(cons h ys) (append h (concat ys))]
      )))
```

Using `foldr`

```
;; [num] -> num
(define sumL
  (lambda (xs)
    (foldr + 0 xs)
  ))
;; [num] -> bool
(define allEven
  (lambda (xs)
    (foldr and #t xs))
  ))
;; [[a]] -> [a]
(define concat
  (lambda (xs)
    (foldr append '() xs)
  ))
```

Defining `foldr`

```
;; (a -> b -> b) -> b -> [a] -> b
(define foldr
  (lambda (f a xs)
    (match xs
      ['() a]
      [(cons h ys) (f h (foldr f a ys))]
      )))
```

```
foldr f a '(x1 x2 ... xn))
```

may be informally understood as

```
(f x1 (f x2 (... (f xn a))))
```

Summary

- Programs in Scheme are expressions
- Expressions may be atomic/values (evaluate to themselves) or composite
- Expressions denote values
- The value denoted by an expression is obtained by evaluation
- Evaluation is follows the call-by-value strategy
- Higher-order functions provide a powerful mechanism for code abstraction and reuse
- Sample higher-order functions seen: `filter`, `map`, `foldr`