Introduction to Scheme

Vitaly Shmatikov

Reading Assignment

- Mitchell, Chapter 3
- "Why Functional Programming Matters" (linked from the course website)
- Take a look at Dybvig's book (linked from the course website)

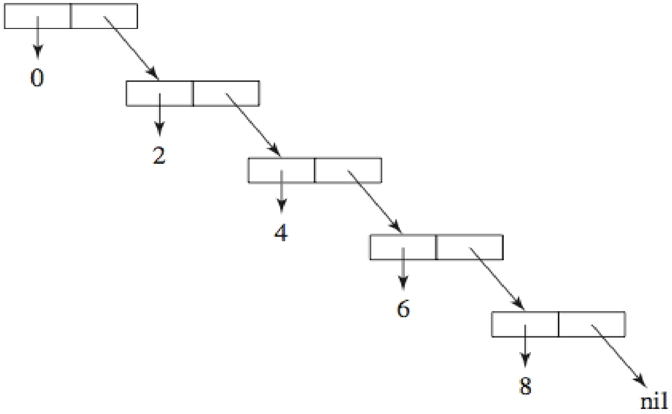
Scheme

- Impure functional language
- Dialect of Lisp
  - Key idea: symbolic programming using list expressions and recursive functions
  - Garbage-collected, heap-allocated (we'll see why)
- Some ideas from Algol
  - Lexical scoping, block structure
- Some imperative features

Expressions and Lists

- Cambridge prefix notation: (f x1 x2 … xn)
  - (+ 2 2)
  - (+ (* 5 4) (- 6 2)) means 5*4 + (6-2)

- List = series of expressions enclosed in parentheses
  - For example, (0 2 4 6 8) is a list of even numbers
  - The empty list is written ()
- Lists represent both functions and data



Elementary Values

- Numbers
  - Integers, floats, rationals
- Symbols
  - Include special Boolean symbols #t and #f
- Characters
- Functions
- Strings
  - "Hello, world"
- Predicate names end with ?
  - (symbol? '(1 2 3)), (list? (1 2 3)), (string? "Yo!")

Top-Level Bindings

- define establishes a mapping from a symbolic name to a value in the current scope
  - Think of a binding as a table: symbol $\rightarrow$ value
  - (define size 2) ; size = 2
  - (define sum (+ 1 2 3 4 5)) ; sum = (+ 1 2 3 4 5)
- Lambda expressions
  - Similar to “anonymous” functions in ML
  - Scheme: (define square (lambda (x) (* x x)))
  - ML: fun square = fn(x) $\Rightarrow$ x*x
    - What’s the difference? Is this even valid ML? Why?

Functions

- (define (name arguments) function-body)
  - (define (factorial n)
    (if (< n 1) 1 (* n (factorial (- n 1)))))
  - (define (square x) (* x x))
  - (define (sumsquares x y)
    (+ (square x) (square y)))
  - (define abs (lambda (x) (if (< x 0) (- 0 x) x)))
- Arguments are passed by value
  - Eager evaluation: argument expressions are always evaluated, even if the function never uses them
  - Alternative: lazy evaluation (e.g., in Haskell)

Expression Evaluation

- Read-eval-print loop
- Names are replaced by their current bindings
  - x ; evaluates to 5
- Lists are evaluated as function calls
  - (+ (* x 4) (- 6 2)) ; evaluates to 24
- Constants evaluate to themselves.
  - ‘red ; evaluates to ‘red
- Innermost expressions are evaluated first
  - (define (square x) (* x x))
  - (square (+ 1 2)) ⇒ (square 3) ⇒ (* 3 3) ⇒ 9

Equality Predicates

- eq? - do two values have the same internal representation?
- eqv? - are two numbers or characters the same?
- equal? - are two values structurally equivalent?
- Examples
  - (eq ‘a ‘a) ⇒ #t
  - (eq 1.0 1.0) ⇒ #f (system-specific) (why?)
  - (eqv 1.0 1.0) ⇒ #t (why?)
  - (eqv “abc” “abc”) ⇒ #f (system-specific) (why?)
  - (equal “abc” “abc”) ⇒ #t

Operations on Lists

- car, cdr, cons
  - (define evens '(0 2 4 6 8))
  - (car evens) ; gives 0
  - (cdr evens) ; gives (2 4 6 8)
  - (cons 1 (cdr evens)) ; gives (1 2 4 6 8)
- Other operations on lists
  - (null? '()) ; gives #t, or true
  - (equal? 5 '(5)) ; gives #f, or false
  - (append '(1 3 5) evens) ; gives (1 3 5 0 2 4 6 8)
  - (cons '(1 3 5) evens) ; gives ((1 3 5) 0 2 4 6 8)
    - – Are the last two lists same or different?

Conditionals

- General form

  (cond (p1 e1) (p2 e2) ... (pN eN))

  - Evaluate p_i in order; each p_i evaluates to #t or #f
  - Value = value of e_i for the first p_i that evaluates to #t or e_N if p_N is "else" and all p_1 ... p_{N-1} evaluate to #f

- Simplified form

  - (if (< x 0) (- 0 x)) ; if-then
  - (if (< x y) x y) ; if-then-else

- Boolean predicates:

  (and (e1) ... (eN)), (or (e1) ... (eN)), (not e)

Other Control Flow Constructs

- **Case selection**
  - (case month

```
((sep apr jun nov)  30)
((feb)          28)
(else           31)
)
```

- **What about loops?**
  - Iteration $\leftrightarrow$ Tail recursion
  - Scheme implementations must implement tail-recursive functions as iteration

Delayed Evaluation

- Bind the expression to the name as a literal...
  - (define sum '(+ 1 2 3))
  - sum ⇒ (+ 1 2 3)
    - – Evaluated as a symbol, not a function
- Evaluate as a function
  - (eval sum) ⇒ 6
- No distinction between code (i.e., functions) and data – both are represented as lists!

Imperative Features

- Scheme allows imperative changes to values of variable bindings
  - (define x `(1 2 3))
  - (set! x 5)
- Is it Ok for new value to be of a different type? Why?
- What happens to the old value?

Let Expressions

- Nested static scope
- (let ((var1 exp1) ... (varN expN)) body)

```
(define (subst y x alist)
      (if (null? alist) '()
          (let ((head (car alist)) (tail (cdr alist)))
                (if (equal? x head)
                      (cons y (subst y x tail))
                      (cons head (subst y x tail)))))
```

- This is just syntactic sugar for a lambda application (why?)

Let*

- (let* ((var1 exp1) ... (varN expN)) body)
  - Bindings are applied sequentially, so var_i is bound in exp_{i+1} ... exp_N
- This is also syntactic sugar for a (different) lambda application (why?)
  - (lambda (var1) (

    (lambda (var2) (... (

    (lambda (varN) (body)) expN) ...) exp1

Functions as Arguments

```
(define (mapcar fun alist)
        (if (null? alist) '()
                (cons (fun (car alist))
                        (mapcar fun (cdr alist)))
))

(define (square x) (* x x))
```

What does (mapcar square '(2 3 5 7 9)) return?

(4 9 25 49 81)

"Folding" a Data Structure

- **Folding:** processing a data structure in some order to construct a return value
  - Example of higher-order functions in action
- Summing up list elements (left-to-right)
  - (foldl + 0 '(1 2 3 4 5)) ⇒ 15
    - Evaluates as (+ 5 (+ 4 (+ 3 (+ 2 (+ 1 0))))). Why?
  - (define (sum lst) (foldl + 0 lst))
- Multiplying list elements (right-to-left)
  - (define (mult lst) (foldr * 1 lst))
  - (mult '(2 4 6)) ⇒ (* (* (* 6 4) 2) 1)) ⇒ 48

Using Recursion

- Compute length of the list recursively
  - (define length
            (lambda(lst)
                    (if (null? lst) 0 (+ 1 (length (cdr list))))))
- Compute length of the list using foldl
  - (define length
            (lambda(lst)
                    (foldl (lambda (_ n) (+ n 1)) 0 lst)
)
)

Ignore 1st argument. Why?

Key Features of Scheme

- Scoping: static
- Typing: dynamic (what does this mean?)
- No distinction between code and data
  - Both functions and data are represented as lists
  - Lists are first-class objects
    - Can be created dynamically, passed as arguments to functions, returned as results of functions and expressions
  - This requires heap allocation (why?) and garbage collection (why?)
  - Self-evolving programs