Announcements

- HW4 due today. There is no HW5. (It was merged into HW4.)
- HW6 (Scheme, Problem 1) out today, due October 23rd
  - We’ll cover 100% of what you need today!
- Rainbow grades available
  - HW1-3, Quiz1-3, Exam1

Last Class

- Static semantic analysis
  - Attribute grammars
- Synthesized and inherited attributes
  - S-attributed grammars
  - L-attributed grammars

Today’s Lecture Outline

- Functional programming languages
- Scheme
  - S-expressions and lists
    - cons, car, cdr
  - Defining functions
  - Examples of recursive functions
    - Shallow recursion, Deep recursion
  - Equality testing
- Higher order functions
- Binding with let and let*

Functional Programming with Scheme

Read: Scott, Chapter 11

Racket/PLT Scheme/DrScheme

- Download Racket (was PLT Scheme (was DrScheme))
  - http://racket-lang.org/
  - Run DrRacket, choose language R5RS
- Online tutorial:
  - Teach yourself Scheme in Fixnum days:
    - http://www.ccs.neu.edu/home/dorai/t-y-scheme/t-y-scheme.html

First, Imperative Languages

- The concept of assignment is central
  - X:=5; Y:=10; Z:=X+Y; W:=f(Z);
- Side effects on memory
- Program semantics (i.e., how the program works): state-transition semantics
  - A program is a sequence of assignment statements with effect on memory (i.e., state)
    1. C := 0;
    2. for I := 1 step 1 until N do
    3.  C := C + a[I]*b[I];
Imperative Languages

- **Functions** (typically called “procedures” or “subroutines”) have side effects:
  - Roughly:
    - A function call affects visible state; i.e., a function call affects execution of other functions, and in general, function calls cannot be replaced by result
    - Also, result of a function call depends on visible state; i.e., function call is not independent of the context of the call

Functional Languages

- **Program semantics: reduction semantics**
  - A program is a set of function definitions and their application to arguments

```
Def IP = (Insert +) º (ApplyToAll *) º Transpose
```

```
IP <<1,2,3>,<6,5,4>> is
  (Insert +) ((ApplyToAll *) ((Transpose
    <<1,2,3>,<6,5,4>>)
    (ApplyToAll *) <<1,6>,<2,5>,<3,4>>)
  (Insert +) <6,10,12>
```

Functional Languages

- In pure functional languages, there is no notion of assignment, no notion of state
  - Variables are bound to values only through parameter associations
  - No side effects!
  - **Referential transparency**
  - Roughly:
    - Result of function application is independent of context where the function application occurs; function application can be replaced by result

Lisp and Scheme

- **Lisp** is the second-oldest high-level programming language!
  - Simple syntax!
  - Program code and data have same syntactic form
    - The S-expression
      - Function application written in prefix form
        - (e1 e2 e3 ... ek) means
          - Evaluate e1 to a function value
          - Evaluate each of e2, ..., ek to values
          - Apply the function to these values
          - (+ 1 3) evaluates to 4
History

Lisp 1950's
John McCarthy

Scheme 1975
Guy Steele
Gerald Sussman

Common Lisp

dynamic scoping
lexical scoping
functions as first class values

Why Scheme?

- Simple! Great to introduce core functional programming concepts
- Reduction semantics
- Lists and recursion
- Higher order functions
- Evaluation order
- Parametric polymorphism
- Later we’ll tackle Haskell and new concepts
- Algebraic data types and pattern matching
- Lazy evaluation
- Type inference

S-expressions

S-expr ::= Name | Number | \{ S-expr \}

- Name is a symbolic constant (a string of chars which starts off with anything that can’t start a Number)
- Number is an integer or real number
- List of zero or more S-expr’s
- E.g., (a (b c) (d)) is a list S-expr

List Functions

- **car** and **cdr**
  - Given a list, they decompose it into first element, rest-of-list portions
  - E.g., **car** of (a (b c) (d)) is a
  - E.g., **cdr** of (a (b c) (d)) is ((b c) (d))
- **cons**
  - Given an element and a list, **cons** builds a new list with the element as its **car** and the list as its **cdr**
  - **cons** of a and (b) is (a b)
- () is the empty list

Questions

```
((a) b (c d))
```

- (car '((a) b (c d))) yields?
- (car '((a) b (c d))) yields?
- (cdr '((a) b (c d))) yields?
- (caddr '((a) b (c d))) yields?

Can compose these operators in a short-hand manner. Can reach arbitrary list element by composition of car’s and cdr’s. (car (cadr (caddr '((a) b (c d))))) can also be written (caddr '((a) b (c d)))

```(a)
```

- (car '((a) b (c d)))
- (car (cdr (cadr '((a) b (c d))))) = (car (cdr '((a) b (c d)))) = (car '((c d))) = (c d)

Quoting

- ‘ or **quote** prevents the Scheme interpreter from evaluating the argument

(quote (+ 3 4)) yields (+ 3 4)

’(+ 3 4) yields (+ 3 4)

Whereas (+ 3 4) yields 7

- Why do we need quote?
Questions

- Recall cons
  - E.g., \((\text{cons } 'a \ ' (b \ c))\) yields \((a \ b \ c)\)

\((\text{cons } 'd \ ' (e))\) yields ?
\((\text{cons } '(a \ b) \ ' (c \ d))\) yields ?
\((\text{cons } '(a \ b \ c) \ ' ((a \ b \ (c \ d)))\) yields ?

Type Predicates

- Note the quote: it prevents evaluation of the argument

\((\text{symbol? } '\text{sam})\) yields \#t
\((\text{symbol? } 1)\) yields \#f
\((\text{number? } '\text{sam})\) yields \#f
\((\text{number? } 1)\) yields \#t
\((\text{list? } '(a \ b))\) yields \#t
\((\text{list? } '\text{sam})\) yields \#f
\((\text{null? } '())\) yields \#t
\((\text{null? } '(a \ b))\) yields \#f
\((\text{zero? } 0)\) yields \#t
\((\text{zero? } 1)\) yields \#f

Can compose these.
\((\text{zero? } (- \ 3 \ 3))\) yields \#t. Note that since this language is fully parenthesized, there are no precedence problems in the expressions!

Question

- What is the typing discipline in Scheme?
  - Static or dynamic?

  Answer: Dynamic typing. Variables are bound to values of different types at runtime. All type checking done at runtime.

Scheme: Defining Functions

\(\text{Fcn-def} ::= (\text{define } \{\text{Fcn-name}\} \{\text{Param}\}) \ S-expr\)
Fcn-name should be a new name for a function. Param should be variable(s) that appear in the S-expr which is the function body.

\(\text{Fcn-def} ::= (\text{define } \{\text{Fcn-name}\} \ Fcn-value)\)
\(\text{Fcn-value} ::= (\text{lambda} \ {'|\text{Param}|}) \ S-expr\)
where Param variables are expected to appear in the S-expr; called a \texttt{lambda} expression.

Examples

\((\text{define } \{\text{zerocheck?}\} \ x)\)
\((\text{if } (= x 0) \ #t \ #f)\)
If-expr ::= \{if \ S-expr0 \ S-expr1 \ S-expr2 \}
where S-expr0 must evaluate to a boolean value; if that value is true, then the If-expr returns the value of S-expr1, else the value of S-expr2.

\((\text{zerocheck? } 1)\) yields \#f
\((\text{zerocheck? } (* \ 1 \ 0))\) yields \#t
Examples

**Examples**

```
(define (atom? object)
 (not (pair? object)))
```

Here `pair?` is a built-in type predicate. It yields `#t` if the argument is a non-trivial S-expr (i.e., something you can take the `cdr` of). It yields `#f` otherwise.

`not` is the built-in logical operator.

What does `atom?` do?

---

**Examples**

```
(define square (lambda (n) (* n n)))
```

associates the Fcn-name `square` with the function value `(lambda (n) (* n n))`.

Lambda calculus is a formal theory of functions.

- Set of functions definable using lambda calculus (Church 1941) is same as set of functions computable as Turing Machines (Turing 1930’s).

---

**Trace of Evaluation**

```
(define (atom? object)
 (not (pair? object)))
```

```
(atom? `(a))
```

- Obtain function value corresponding to `atom?`
  - Evaluate `(a)` obtaining `(a)`
- Obtain function value corresponding to `not`
  - Evaluate `(pair? `(a))`
  - Obtain function value corresponding to `pair?`
    - Evaluate `(a)` obtaining `(a)`
    - Return value `#t`
  - Return `#f`
  - Return `#f`

---

**Read-eval-print Loop**

- Scheme interpreter runs read-eval-print loop

  - **Read** input from user
    - A function application
  - **Evaluate** input
    - `(e1 e2 e3 ... ek)`
      - Evaluate `e1` to obtain a function
      - Evaluate `e2`, `...`, `ek` to values
      - Execute function body using values from previous step as parameter values
      - Return value
  - **Print** return value

---

**Conditional Execution**

```
(if e1 e2 e3)
```

```
(cond (el h1) (e2 h2) ... (en-1 hn-1) (else hn))
```

Cond is like if – then – else if construct

```
(define (zerocheck? x)
 (cond ((= x 0) #t) (else #f))
```

OR

```
(define (zchk? x)
 (cond ((number? x) (zero? x))
   (else #f))
```

---
Recursive Functions

(define (len x)
  (cond ((null? x) 0) (else (+ 1 (len (cdr x))))))

(len '(1 2)) should yield 2.
Trace: (len `(1 2)) -- top level call
  x = (1 2)
    (len `2) -- recursive call 1
      x = (2)
        (len `()) -- recursive call 2
          x = ()
            returns 0 -- return for call 2
            returns (+ 1 0) = 1 -- return for call 1
    returns (+ 1 1) = 2 -- return for top level call
(len `'(a b (c d))) yields what?

Recursive Functions

(define (app x y)
  (cond ((null? x) y) ((null? y) x)
        (else (cons (car x) (app (cdr x) y)))))

app is a shallow recursive function

What does app do?

(app '(()) '()) yields ?
(app '(()) '(1 4 5)) yields ?
(app '(5 9) '(a (4) 6)) yields ?

Exercise

(define (len x)
  (cond ((null? x) 0) (else (+ 1 (len (cdr x))))))

Write a version of len that uses if instead of cond

Write a function countlists that counts the number of list elements in a list. E.g.,

(countlists '(a)) yields 0
(countlists '(a (b c (d) (e))) yields 2

Recall (list? 1) returns true if 1 is a list, false otherwise

fun counts atoms in a list

(define (fun x)
  (cond ((null? x) 0)
        ((atom? x) 1)
        (else (+ (fun (car x)) (fun (cdr x))))))

fun is a deep recursive function

fun counts atoms in a list

(define (atomcount x)
  (cond ((null? x) 0)
        ((atom? x) 1)
        (else (+ (atomcount (car x)) (atomcount (cdr x))))))

atomcount is a deep recursive function

(atomcount '(a)) yields 1
(atomcount '(1 (2 (3)) (5))) yields 4

Trace: (atomcount '(1 (2 (3)))
  1> (+ (atomcount 1) (atomcount '(2 (3)))))
  2> (+ (atomcount 2) (atomcount '(3)))
    3> (+ (atomcount 3) (atomcount '(3)))
      4> (+ (atomcount '(3)) (atomcount '1))
        5> (+ (atomcount 3) (atomcount '1))

Group Exercise

Write a function flatten that flattens a list

(flatten '(1 (2 (3)))) yields (1 2 3)
Lecture Outline

- Functional Programming Languages
- Scheme
  - S-expressions and lists
  - cons, car, cdr
  - Defining functions
  - Examples of recursive functions
    - Shallow recursion, Deep recursion
  - Equality testing
  - Higher order functions
  - Binding with `let` and `let*`

Equality Testing

`eq?`
- Built-in predicate that can check atoms for equal values
- Doesn’t work on lists as you might expect!

`eql?`
- Our predicate that works on lists
  
  ```scheme
  (define (eql? x y)
    (or (and (atom? x) (atom? y) (eq? x y))
        (and (not (atom? x)) (not (atom? y))
             (eql? (car x) (car y))
             (eql? (cdr x) (cdr y))))
  ```

`equal?`
- Built-in predicate that works on lists

Examples

- `(eql? '(a) '(a))` yields what?
- `(eql? 'a 'b)` yields what?
- `(eql? '(a) '(a))` yields what?
- `(eq? 'a 'a)` yields what?
- `(eq? '(a) '(a))` yields what?

Models for Variables

- Value model for variables
  - A variable is a location that holds a value
  - I.e., a named container for a value
  - `a := b`

- Reference model for variables
  - A variable is a reference to a value
  - Every variable is an l-value
  - Requires dereference when r-value needed (usually, but not always implicit)

Models for Variables: Example

```
(define b 2)
(define c b)
(define a (+ b c))
```

Equality Testing: How does `eq?` work?

- Scheme uses the reference model for variables!
- `(define (f x y) (list x y))`
- Call `(f 'a 'a)` yields `(a a)`
- `eq?` checks that `x` and `y` both point to the same place.
- Call `(f '(a) '(a))` yields `'(a) '(a)`
- `x` and `y` do not refer to the same list.
Models for Variables

- C/C++, Pascal, Fortran
  - Value model
- Java
  - Mixed model: value model for simple types, reference model for class types
- JS, Python, R, etc.
  - Reference model
- Scheme
  - Reference model! `eq?` is “reference equality” (akin of Java’s `==`), `equal?` is value equality