Announcements

- HW4 due today

- HW3 and Quiz 4 are being graded, grades will be up tomorrow
  - If you have questions, let us know!

- HW5 (Scheme, Problem 1) out today, due March 28th

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 10

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** have side effects:
  - Roughly:
    - Function application affects visible state; i.e., function application affects execution of other functions, and in general, function application cannot be replaced by result
    - Also, the result of a function application depends on the visible state; i.e., function application is not independent of the context of application

**Functional Languages**

- Functions are usually not first-class values
  - A first-class value is one that can be passed as argument and returned as value from functions
    - In a language with assignments, can be assigned into a variable or structure
  - Are functions in C first-class values?

- Imperative languages can be compiled
- Imperative languages can use manual memory management
- Imperative languages have complex syntax

**Program semantics: reduction semantics**

A program is a set of function definitions and their application to arguments

\[
\text{Def } \text{IP} = (\text{Insert } +)^6 (\text{ApplyToAll } *)^{n} \text{ Transpose}
\]

\[
\text{IP} <<1,2,3>,<6,5,4>> \quad \text{is}
\]
\[
(\text{Insert } +) ((\text{ApplyToAll }*) (\text{Transpose}
\quad <<1,2,3>,<6,5,4>>))
\]
\[
(\text{Insert } +) ((\text{ApplyToAll }*) <<1,6>,<2,5>,<3,4>>)
\]
\[
(\text{Insert } +) <6,10,12>
\]

28

**Functional Languages**

- Functions are first-class values
  - Can be returned as value of a function application
  - Can be passed as an argument
  - In a language with assignment, can be assigned into variables and structures and saved
  - Unnamed functions exist as values
  - Functional languages are usually interpreted
  - Functional languages have implicit memory management, i.e., garbage collection
  - Functional languages have simple syntax
Functional Languages

- **Lisp** designed for symbolic computing
  - Simple syntax!
  - Program code and data have same syntactic form
  - The S-expression
  - Function application written in prefix form
    \((e_1 e_2 e_3 \ldots e_k)\) means
    - Evaluate \(e_1\) to a function value
    - Evaluate each of \(e_2, \ldots, e_k\) 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

  - dynamic scoping
  - lexical scoping
  - functions as first class values

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

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

- ((a b (c d))
  - (car ‘((a b c)) yields ?
  - (car ‘((a) b (c d))) yields ?
  - (cdr ‘(a b c)) yields ?
  - (cdr ‘((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 (cdr (cdr ‘((a) b (c d)))))
  - can also be written
  - (caddr ‘((a) b (c d)) )
  - (car (cdr (cdr ‘((a) b (c d))))) = (car (cdr ‘((b c d) )\)) = (car ‘((c d)) = (c d)
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 \texttt{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

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 \texttt{let} and \texttt{let*}

Scheme: Defining Functions

\[
\text{Fcn-def} ::= (\text{define} \ \{\text{Fcn-name} \ (\text{Param})\} \ S-expr)
\]

\[\text{Fcn-name should be a new name for a function. Parameter(s) appear in S-expr which is the function body.}\]

\[
\text{Fcn-def} ::= (\text{define} \ \text{Fcn-name} \ \text{Fcn-value})
\]

\[\text{Fcn-value} ::= (\text{lambda} \ \{\text{Param}\} \ S-expr)
\]

\[\text{Param variables are expected to appear in S-expr; called a \texttt{lambda expression.}}\]

Examples

\[
(\text{define} \ \text{(zerocheck?\ x)}
  \begin{cases}
  & (\text{if} \ (= \ x\ 0) \ \#t \ \#f) \\
  \end{cases})
\]

\[\text{If-expr} ::= (\text{if} \ S-expr0 \ S-expr1 \ S-expr2)\]

\[\text{where S-expr0 must evaluate to a boolean value; if that value is true, then the If-expr returns the value of S-expr1, else the value of S-expr2.}\]

\[
(\text{zerocheck?\ 1)\ yields\ \#f.}
\]

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

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

Here \texttt{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 \texttt{cdr} of). It yields \#f otherwise.

\texttt{not} is the built-in logical operator.

What does \texttt{atom?} do?

---

Examples

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

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

---

Read-eval-print Loop

- Scheme interpreter runs \texttt{read-eval-print} loop.

  - \texttt{Read} input from user
    - A function application
  
  - \texttt{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 of function
  
  - \texttt{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

A recursive function `len` is defined as:

```
(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

Exercise

A version of `len` that uses `if` instead of `cond` is:

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

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

- `(countlists '(a)) yields 1`
- `(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**

A function `atomcount` counts atoms in a list:

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

- `(atomcount '(1)) yields 1`
- `(atomcount '(1 (2 (3))) (5)) yields 4`

**Group Exercise**

Write a function `flatten` that flattens a list:

```
(define (flatten x)
  (cond ((null? x) x)  (else (cons (car x) (flatten (cdr x))))))
```

(Spring 16 CSCI 4430, A Milanova/BG Ryder)

- `(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

  l-value (the location) r-value (the value held in that location)

- 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

b := 2;
  c := b;
  a := b + c;

- Value model for variables
  - b := 2
  - c := b
  - a := b + c

- Reference model for variables
  - b := 2
  - c := b
  - a := b + c

Equality Testing: How does eq? work?

- Scheme uses the reference model for variables!

```scheme
(define (f x y) (list x y))
```

Call (f 'a 'a) yields (a a)
x refers to atom a and y refers to atom 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.