**Function Application**

- A function application is written like this:

  \[(\text{operator } \text{arg}_1 \text{ arg}_2 \ldots \text{ arg}_n)\]

- The evaluation proceeds as follows:
  1. Evaluate \(\text{operator}\). The result should be a function \(\mathcal{F}\).
  2. Evaluate
      \[\text{arg}_1, \text{arg}_2, \ldots, \text{arg}_n\]
      to get
      \[\text{val}_1, \text{val}_2, \ldots, \text{val}_n\]
  3. Apply \(\mathcal{F}\) to \(\text{val}_1, \text{val}_2, \ldots, \text{val}_n\).

**Function Application — Examples**

> (+ 4 5)  
9  
> (+ (+ 5 6) 3)  
14  
> 7  
7  
> (4 5 6)  
eval: 4 is not a function  
> #t  
#t

**Background**

- Scheme is based in LISP which was developed by John McCarthy in the mid 50s.
- Functions and data share the same representation: S-Expressions.
- A basic LISP implementation needs six functions \(\text{cons}, \text{car}, \text{cdr}, \text{equal}, \text{atom}, \text{cond}\).
- Scheme was developed by Sussman and Steel in 1975.
List Functions — Examples

> (memq 'z '(x y z w))
#t
> (car (cdr (car '(((a) b (c d))))))
(c d)
> (caddr '(((a) b (c d))))
(c d)
> (cons 'a '())
(a)
> (cons 'd '(e))
(d e)
> (cons '(a b) '(c d))
((a b) (c d))

Predicates

> (list? (+ 5 6)) #f
> (list? '(+ 5 6)) #f
> (list? '(+ 5 6)) #t
> (number? 5) #t
> (number? 'hello) #f
> (zero? 5) #f
> (> 5 9) #t
> (< 5 9) #t
> (boolean #t) #t
> (string "hello") #t

List Functions

eq? Are two atoms the same?
null? Is the list empty?
car Return the first element of a list.
cdr Return the rest of a list when the first element has been removed.
(cons frst rest) Return a list consisting of frst as the first element and rest as the rest of the list.
(quote L) Don’t evaluate L. Also ’L.

List Functions — Examples

> (car '(2 3 4))
2
> (cdr '(2 3 4))
(3 4)
> (cons 2 '(3 4))
(2 3 4)
> (cdr '(2))
()
> (cdr '(((a b (c d))))
(b (c d))
> (null? '()) #t
Lambda Expressions

- A lambda-expression evaluates to a function:

\[
\text{lambda (x) (* x x)}
\]

- \( x \) is the function's formal parameter.
- Lambda-expressions don't give the function a name.
- Evaluating the function:

\[
> ((\text{lambda (x) (* x x)}) 3)
\]

Let Expressions

- A let-expression binds names to values:

\[
\text{let ((name value1) (name2 value2) ...)} expression
\]

- The first argument to let is a list of (name value) pairs.
- The second argument is the expression to evaluate.

\[
> \left(\text{let ((a 3) (b 4) (square (lambda (x)(* x x))) (plus +))}
\right. \\
> \left.\left(\text{sqrt (plus (square a) (square b))})\right)\right)
\]

Defining Names

- define binds an expression to a global name:

\[
\text{(define (name arg1 arg2 ...)} expression)
\]

Conditionals

- if-expressions:

\[
\text{(if (condition then else)}
\]

- cond-expressions:

\[
\text{(cond (cond1 res1) (cond2 res2) ...)(else else)}
\]

Let Expressions

<table>
<thead>
<tr>
<th>Slide 4-9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Defining Names</td>
</tr>
<tr>
<td>• define binds an expression to a global name:</td>
</tr>
</tbody>
</table>
| \[
\text{(define (name arg1 arg2 ...)} expression)
\] |
| > (define (f) ’hello) |
| > (f) |
| hello |
| > (define (square x) (* x x)) |
| > (square 3) |
| 9 |
| Conditionals |
| • if-expressions: |
| \[
\text{(if (condition then else)}
\] |
| • cond-expressions: |
| \[
\text{(cond (cond1 res1) (cond2 res2) ...)(else else)}
\] |
| > (if (< 2 3) 4 5) |
| 4 |
| > (cond ((< 2 3) 4) ((= 2 3) 5) (else 6)) |
| 4 |
| Lambda Expressions |
| • A lambda-expression evaluates to a function: |
| \[
\text{lambda (x) (* x x)}
\] |
| \( x \) is the function's formal parameter. |
| • Lambda-expressions don’t give the function a name. |
| • Evaluating the function: |
| \[
> ((\text{lambda (x) (* x x)}) 3)
\] |
| 9 |
| Slide 4-10 |
| Slide 4-8 |
| Let Expressions |
| • A let-expression binds names to values: |
| \[
\text{let ((name value1) (name2 value2) ...)} expression
\] |
| • The first argument to let is a list of (name value) pairs. |
| • The second argument is the expression to evaluate. |
| > (let ((a 3) (b 4) (square (lambda (x)(* x x))) (plus +)) |
| > (sqrt (plus (square a) (square b)))) |
| 5.0 |
Recursive Functions...

- Count the number of atoms in an S-expression.
- This is called car-cdr-recursion.

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

> (atomcount '(1 (2 3 4) (5)))
5
```

Recursive Functions...

- Compute the length of a list.
- This is called cdr-recursion.

```
(define (abs-val x) (cond ((>= x 0) x) (else (- x))))
(define (abs-list l)
    (cond ((null? l) '())
         (else (cons (abs-val (car l))
                      (abs-list (cdr l))))))

> (abs-list '(1 2 -3 -4 5)))
(1 2 3 4 5)
```

Recursive Functions...

- Sum the integers from 0 to n.

```
(define (sum-n n)
    (cond ((= n 0) 0)
         (else (+ n (sum-n (- n 1))))))

> (sum-n 6)
21
```

Recursive Functions...

- Map a list of numbers to a new list of their absolute values.

```
(define (abs-count x)
    (cond ((null? x) 0)
         ((atom? x) 1)
         (else (+ (abs-count (car x))
                  (abs-count (cdr x))))))
```

```bash
> (abs-count '(1 (2 3) (4)))
5
```
Append

\begin{verbatim}
(define (append L1 L2)
  (cond ((null? L1) L2)
        (else (cons (car L1)
                    (append (cdr L1) L2)))))
\end{verbatim}

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

> (append '() '(3 4))
(3 4)

> (append '(1 2) '())
(1 2)

Deep vs. Shallow Equality

\begin{itemize}
  \item (eq? A B) checks if its two arguments are the same.
  \item (equal? A B) checks if its two arguments have the same structure.
\end{itemize}

> (eq? (cons 'a () ) (cons 'a '()))
#f
> (equal? '(1 2 3) '(1 2 3))
#t
Higher-Order Functions: map

- The builtin function

\[(\text{map } f \ L)\]
constructs a new list by applying the function \(f\) to every element of the list \(L\).

\[(\text{map } f \ '(e_1 \ e_2 \ e_3 \ e_4)) \Rightarrow ((f \ e_1) \ (f \ e_2) \ (f \ e_3) \ (f \ e_4))\]

- \text{map} is a higher-order function, i.e. it takes another function as an argument.

\[>\ (\text{map } (\lambda x \ (+ \ 1 \ x)) \ '(-1 \ 2 \ -3))\]
\[\ (0 \ 3 \ -2)\]
\[>\ (\text{map } \text{cons} \ '((a \ b \ c)) \ '((1) \ (2) \ (3)))\]
\[\ ((a \ 1) \ (b \ 2) \ (c \ 3))\]

Higher-Order Functions: apply

- The builtin function

\[(\text{apply } f \ L)\]
folds a list together by successively applying the function \(f\) to the elements of the list \(L\).

\[(\text{apply } f \ '(e_1 \ e_2 \ e_3 \ e_4)) \Rightarrow (f \ e_1 \ (f \ e_2 \ (f \ e_3 \ e_4)))\]

\[>\ (\text{apply } + \ '(1 \ 2 \ 3))\]
\[\ 6\]
\[>\ (\text{map } \text{append} \ '((a) \ (b) \ (c)))\]
\[\ (a \ b \ c)\]

Example: Binary Trees

- A binary tree can be represented as nested lists:

\[(4 \ (2 \ () \ () \ (6 \ (5 \ () \ () \ () \ ())))]\n
- Each node is represented by a triple

\[(\text{data} \ \text{left-subtree} \ \text{right-subtree})]\n
- Empty subtrees are represented by ()

Example: Binary Trees...

\[(\text{define} \ (\text{key} \ \text{tree}) \ (\text{car} \ \text{tree}))\]
\[(\text{define} \ (\text{left} \ \text{tree}) \ (\text{cadr} \ \text{tree}))\]
\[(\text{define} \ (\text{right} \ \text{tree}) \ (\text{caddr} \ \text{tree}))\]

\[(\text{define} \ (\text{print-spaces} \ N)\]
\[\quad (\text{cond} \ ((= \ N \ 0))\]
\[\quad \hspace{1cm} (\text{else})\]
\[\quad \hspace{1.5cm} (\text{display} \ "\")\]
\[\quad \hspace{1.5cm} (\text{print-spaces} \ (- \ N \ 1))))))\]

\[(\text{define} \ (\text{print-tree} \ \text{tree})\]
\[\quad (\text{print-tree-rec} \ \text{tree} \ 0))\]
References...

- Language reference manual: http://www.swiss.ai.mit.edu/ftpdir/scheme-reports/r5rs.ps


Homework

- Read Scott, pp. 594–605.

- Read up on Scheme using one of the online tutorials/manuals or the language reference manual.

- Write a function swapFirstTwo which swaps the first two elements of a list. Example: (1 2 3 4) ⇒ (2 1 3 4).

- Write a function swapTwoInLists which, given a list of lists, forms a new list of all elements in all lists, with first two of each swapped. Example: ((1 2 3) (4) (5 6)) ⇒ (2 1 3 4 5 6).

References


- Tutorials:
  - http://cs.wwc.edu/%7Ecs_dept/KU/PR/Scheme.html
  - http://www.cis.upenn.edu/%7Eungar/CIS520/scheme-tutorial.html

- http://dmoz.org/Computers/Programming/Languages/Lisp/Scheme