ECE570 Lecture 1: Recursion and Lists

Jeffrey Mark Siskind

School of Electrical and Computer Engineering

Fall 2013

PURDUE UNIVERSITY

Course Overview—I

- What course this is.
- When and where it meets.
- The course staff.
- Who am I.
- My office hours, office, email, phone.
- I've taught this course twenty eight times before.

Course Overview—II

- Two dimensional matrix

  problems: diagnosis, puzzles, games, scene interpretation, planning, . . .
  methods: nondeterministic search, constraints, logic, lifting, . . .
- Learn by doing

Course Overview—III

- Course texts
- `http://engineering.purdue.edu/~ee473`
    - Syllabus, Lectures, Problem Sets, Documentation
- Attend every lecture
- Prerequisites
- Computer accounts
- Course software
- Problem sets
- Grading
- Collaboration
- Party

Course Overview—IV

- Communication
  1. Class
  2. `http://engineering.purdue.edu/~ee473`
     - *What's New?*
  3. Email
     - `ee473-staff-list@ecn.purdue.edu`,
     - `ee473-students-list@ecn.purdue.edu`
  4. Office hours
  5. Phone
  6. Openness Policy

Course Overview—V

In the event of a major campus emergency, course requirements, deadlines and grading percentages are subject to changes that may be necessitated by a revised semester calendar or other circumstances. Such changes will be announced to the course email mailing list.

Scheme

SCHEME programs are built out of *expressions*.

Expressions are *evaluated*.

$\Longrightarrow$ means *evaluates to*.

`(+ 1 2)` $\Longrightarrow$ `3`

`(- 1 2)` $\Longrightarrow$ `-1`

`(+ (* 2 3) 4)` $\Longrightarrow$ `10`

`(/ 10 5)` $\Longrightarrow$ `2`

REPL: Read-Eval-Print Loop

Scheme is an interactive language.

```
Scheme->C -- 15mar93jfb
> (+ 1 2)
3
> (- 1 2)
-1
> (+ (* 2 3) 4)
10
> (/ 10 5)
2
>
```

Definitions—I

```
> (sqrt (+ (sqr (- 4 3)) (sqr (- 9 8))))
1.4142135623731
> (sqrt (+ (sqr (- 8 2)) (sqr (- 9 8))))
6.08276253029822
> (define (distance x1 y1 x2 y2)
   (sqrt (+ (sqr (- x2 x1)) (sqr (- y2 y1)))))
> (distance 3 8 4 9)
1.4142135623731
> (distance 2 8 8 9)
6.08276253029822
>
```

Definitions—II

Put in a file then load with `c-z l`

```
(define (factorial n)
 (if (= n 0)
     1
     (* n (factorial (- n 1)))))
```

```
> (factorial 5)
120
```

Recursion as Induction

$$\begin{array}{rcll} 0! & = & 1 & \text{base case} \\ n! & = & n \times (n-1)! & \text{inductive case} \end{array}$$

```
(define (factorial n)
 (if (= n 0)
     1
     (* n (factorial (- n 1)))))
```

Peano Arithmetic—I

$$
\begin{aligned}
n^+ &= n+1 \\
n^- &= n-1 \\
0 &
\end{aligned}
$$

```
(define (increment n) (+ n 1))

(define (decrement n) (- n 1))

0

(zero? n)
```

Peano Arithmetic—II

$$\begin{array}{rcll} m+0 & = & m & \text{base case} \\ m+n & = & (m+n^{-})^{+} & \text{inductive case} \end{array}$$

```
(define (+ m n)
 (if (zero? n)
     m
     (increment (+ m (decrement n)))))
```

Peano Arithmetic—III

$$\begin{array}{rcll} m-0 & = & m & \text{base case} \\ m-n & = & (m-n^{-})^{-} & \text{inductive case} \end{array}$$

```
(define (- m n)
 (if (zero? n)
     m
     (decrement (- m (decrement n)))))
```

Peano Arithmetic—IV

$$\begin{array}{rcll} m \times 0 & = & 0 & \text{base case} \\ m \times n & = & (m \times n^{-}) + m & \text{inductive case} \end{array}$$

```
(define (* m n)
 (if (zero? n)
     0
     (+ (* m (decrement n)) m)))
```

Peano Arithmetic—V

$$\begin{aligned} \frac{0}{n} &= 0 && \text{base case} \\ \frac{m}{n} &= \left(\frac{m-n}{n}\right)^{+} && \text{inductive case} \end{aligned}$$

```
(define (/ m n)
 (if (zero? m)
     0
     (increment (/ (- m n) n))))
```

Peano Arithmetic–VI

$$\begin{array}{rcll} & & m \not< 0 & \text{base case} \\ & & 0 < n & \text{base case} \\ m^- < n^- & \rightarrow & m < n & \text{inductive case} \end{array}$$

```
(define (< m n)
 (if (zero? n)
     #f
     (if (zero? m)
         #t
         (< (decrement m) (decrement n)))))
```

COND Syntax

$$
\left.
\begin{array}{ll}
\texttt{(cond} & \texttt{(}p_1 \;\; e_1\texttt{)} \\
 & \texttt{(}p_2 \;\; e_2\texttt{)} \\
 & \vdots \\
 & \texttt{(}p_n \;\; e_n\texttt{)} \\
 & \texttt{(else} \;\; e\texttt{))}
\end{array}
\right\}
\rightsquigarrow
\left\{
\begin{array}{l}
\texttt{(if } p_1 \\
\quad\quad e_1 \\
\quad\quad \texttt{(if } p_2 \\
\quad\quad\quad\quad e_2 \\
\quad\quad\quad\quad \vdots \\
\quad\quad\quad\quad \texttt{(if } p_n \\
\quad\quad\quad\quad\quad\quad e_n \\
\quad\quad\quad\quad\quad\quad e\texttt{)...))}
\end{array}
\right.
$$

Uses of COND

```
(define (< m n)
 (if (zero? n)
     #f
     (if (zero? m)
         #t
         (< (decrement m) (decrement n)))))
```

```
(define (< m n)
 (cond ((zero? n) #f))
       ((zero? m) #t)
       (else (< (decrement m) (decrement n))))
```

Syntax vs. Semantics

Procedures extend the *semantics* of a language.

Procedures are *evaluated* via $\Longrightarrow$.

Macros extend the *syntax* of a language.

Macros *rewrite* expressions to other expressions.

$\rightsquigarrow$ means *rewrites to*.

SCHEME contains both builtin syntax and builtin semantics.

Users can define new procedures to extend the semantics of SCHEME.

It is possible for the user to define new syntax via macros. We will not have need for this in this course. So I will not teach how. See the manual for details.

Lists—I

```
(list 1 2 3) ⟹ (1 2 3)

(first (list 1 2 3)) ⟹ 1

(rest (list 1 2 3)) ⟹ (2 3)

(first (rest (list 1 2 3))) ⟹ 2

(rest (rest (list 1 2 3))) ⟹ (3)

(rest (rest (rest (list 1 2 3)))) ⟹ ()
```

Lists—II

```
(list) ⟹ ()

(null? (list)) ⟹ #t

(null? (list 1)) ⟹ #f

(null? (list (list))) ⟹ #f

(cons 1 (list 2 3)) ⟹ (1 2 3)

(cons 1 (list)) ⟹ (1)

(cons (list 1 2) (list 3 4)) ⟹ ((1 2) 3 4)
```

List Processing—I

```
(define (length list)
 (if (null? list)
     0
     (+ (length (rest list)) 1)))
```

List Processing—II

```
(define (list-ref list i)
 (if (= i 0)
     (first list)
     (list-ref (rest list) (- i 1))))
```

List Processing—III

```
(define (list-replace-ith list i x)
 (if (= i 0)
     (cons x (rest list))
     (cons (first list)
           (list-replace-ith (rest list) (- i 1) x))))
```

List Processing—IV

```
(define (list-insert-ith list i x)
 (if (= i 0)
     (cons x list)
     (cons (first list)
           (list-insert-ith (rest list) (- i 1) x))))
```

List Processing—V

```
(define (list-remove-ith list i)
 (if (= i 0)
     (rest list)
     (cons (first list)
           (list-remove-ith (rest list) (- i 1)))))
```

List Processing—VI

```
(define (member? x list)
 (cond ((null? list) #f)
       ((= x (first list)) #t)
       (else (member? x (rest list)))))
```

List Processing—VII

```
(define (position x list)
 (cond ((null? list) (panic "Element not found"))
       ((= (first list) x) 0)
       (else (+ (position x (rest list)) 1))))
```

List Processing—VIII

```
(define (remove-one x list)
 (cond ((null? list) (list))
       ((= (first list) x) (rest list))
       (else (cons (first list) (remove-one x (rest list))))))
```

List Processing—IX

```
(define (remove-all x list)
 (cond ((null? list) (list))
       ((= (first list) x) (remove-all x (rest list)))
       (else (cons (first list) (remove-all x (rest list))))))
```

List Processing—X

```
(define (replace-one x y list)
 (cond ((null? list) (list))
       ((= (first list) x) (cons y (rest list)))
       (else (cons (first list) (replace-one x y (rest list))))))
```

List Processing—XI

```
(define (replace-all x y list)
 (cond
  ((null? list) (list))
  ((= (first list) x) (cons y (replace-all x y (rest list))))
  (else (cons (first list) (replace-all x y (rest list))))))
```

List Processing—XII

```
(define (append list1 list2)
 (if (null? list1)
     list2
     (cons (first list1) (append (rest list1) list2))))
```

List Processing—XIII

```
(define (reverse l)
 (if (null? l)
     (list)
     (append (reverse (rest l)) (list (first l)))))
```

List Processing—XIV

```
(define (sum list)
 (if (null? list)
     0
     (+ (first list) (sum (rest list)))))
```

List Processing—XV

```
(define (product list)
 (if (null? list)
     1
     (* (first list) (product (rest list)))))
```