8. Elements of Racket

Robert R. Snapp
snapp@cs.uvm.edu

Department of Computer Science
University of Vermont

Last modified on October 3, 2014
1 Motivation

2 Racket Data Types
   - Constants
   - Symbols
   - Functions
   - Composite Objects: lists, vectors, and pairs

3 Ordinary Functions
   - Special Forms

4 Functions that dissect lists: car and cdr

5 Functions that build lists: list and cons

6 Implementing Trémaux in Racket
Goal: Create a racket program that threads a maze
Representing a graph as a list:

'( (a (b)) (b (a c d)) (c (b)) (d (b e f)) (e (d)) (f (d g h))
  (g (f)) (h (f i j)) (i (h j l)) (j (h i k)) (k (j))
  (l (i m n)) (m (l)) (n (l o p)) (o (n)) (p (n)) )
Details of Racket Data Types

- Constants (or *literals*)
- Symbols (or *identifiers*)
- Functions (or *procedures*)
- Composite Objects:
  - Lists
  - Vectors
  - Pairs
Details of Racket Data Types: Constants

Constants (or literals) are things that evaluate to themselves:

- Numerals: 1, -3.14159, 3.3e32, 3/5
- Characters: \a, A, \space, \newline
- Strings: "Happy Monday!", "a"
- Booleans: #t and #f
Details of Racket Data Types: Symbols

Symbols (or identifiers) are things that evaluate to a bound value

- A symbol can be *any* sequence of letters (a–z, A–Z), digits (0–9), and special characters ? ! . + – * / < > : $ % ^ & _ ~ such that the symbol does not start with a character that can start a number.

- A symbol can be as long as necessary

- Although different case can be used, some implementations make no distinction between them: e.g. `dog`, `Dog`, and `DOG` might represent the same symbol:
  ```racket
  (define dog '(bow wow))
  DOG returns '(bow wow). DrRacket is, however, case sensitive.
  ```

- The *keyword* `define` binds a value to an identifier:
  After evaluating, `(define x 3)`, then, `x ⇒ 3`.

- Once a symbol is defined, its value can be changed with `set!`. Thus, after `(set! x 4)`, then `x` evaluates to 4. (*Caution*: Use `set!` sparingly.)
Details of Racket Data Types: Functions

- Arithmetic functions (e.g., +, -, *, /, etc.) accept and return numerals: e.g.,
  
  (+ 1 2 3) ⇒ 6
  (expt 2 5) ⇒ 32
  (remainder 5 2) ⇒ 1

- Predicates are functions that return either #t or #f, e.g.,
  
  (zero? 0) ⇒ #t
  (zero? 1) ⇒ #f
  (null? '()) ⇒ #t
  (null? '(a b c)) ⇒ #f
  (eq? 'a 'a) ⇒ #t
  (= (* 5 6) 30) ⇒ #t
  (> 2 1) ⇒ #t
  (<= 3 4 4) ⇒ #t
Racket Functions (cont.)

- A function or procedure is created with the special form `define`. For example,
  
  ```racket
  (define (square x)
    (* x x))
  ```

  creates a function that computes the square of its argument. Thus after evaluating the above
  
  `(square 0) ⇒ 0, (square -3) ⇒ 9, (square 1.5) ⇒ 2.25`

- In the above form, `define` accomplishes two tasks in sequence.
  - First, an *anonymous* function that computes the square is created.
  - Second, that anonymous function is given the name “`square`.”

- Anonymous procedures can be constructed using the keyword `lambda`, e.g.
  
  ```racket
  (lambda (x) (* x x))
  ```
  is an anonymous squaring function.

- Alternatively, one can define `square` as follows:
  
  ```racket
  (define square
    (lambda (x)
      (* x x)))
  ```
Details of Racket Data Types: Composite Objects

- A **list** is a sequence of elements surrounded by parentheses, e.g.
  - (apple pear plum)
  - (< 2 7 38)
  - (- "x" (#\b #f)), and
  - () is an empty list.

  (Note that a list may contain other lists.)

- A **vector** is a sequence of elements preceded by #( and followed by )”, e.g.
  - #(1.6 23 -4.2e5), and
  - #(\a \b \c).

- A **pair** is a composite object with exactly two slots that are delimited by a **dot**: e.g.
  - (a . b).

  (Lists are actually constructed implicitly with pairs: the first element is the **car**, the second is the **cdr**.)
Functional Programming

- In racket (and other functional languages), each procedure is expressed as a “mathematical” function that is evaluated during execution. This paradigm is called functional programming.

- Most often functions to be evaluated are implemented as a list. The first item in the list specifies a defined function (or procedure); the remaining items are the function’s arguments:

  \((\text{procedure } \text{arg}_1 \text{ arg}_2 \ldots \text{arg}_n)\).

- For ordinary functions, arguments are evaluated before the procedure. So before the addition can be performed in the expression

  \((+ (\text{sqrt} 9) (\text{expt} 2 10)))\),

  DrRacket will first evaluate the two arguments: \((\text{sqrt} 9)\) becomes 3, and \((\text{expt} 2 10)\) becomes 1024. Subsequently, the expression \((+ 3 1024)\) evaluates to 1027. As in algebra, expressions in parenthesis are evaluated first. Here, in addition, the arguments \(\text{arg}_1, \text{arg}_2, \ldots, \text{arg}_n\) are evaluated before procedure.

- An error occurs if an unbound identifier is evaluated.
Special Forms

Not everything that we would like to do can be accomplished using ordinary functions. Procedures in which the arguments are not evaluated first are called special forms. Some examples are

- `define` and `set!`
- `lambda`,
- `if` and `cond`,
- `quote`,
- `and` and `or`,
- `let` and `let*`
- `letrec`,
- `case`,
- `do`.
Special Forms: if

- The keyword if enables branching, enabling an algorithm to do one of two different operations depending on the outcome of a test. It usually appears in the form

  \[
  \text{(if } \text{test-expression} \\
  \quad \text{then-expression} \\
  \quad \text{else-expression}),
  \]

  where the then-expression is only evaluated if the test-expression evaluates to #t, otherwise the else-expression is evaluated. (More generally, the else-expression is optional.)

Example:

\[
\text{(define (factorial n)} \\
\quad \text{(if (zero? n)} \\
\quad \quad 1 \\
\quad \quad (\text{* n (factorial (- n 1)}))))
\]

What do you do if you want your algorithm to have more than two branches?
Special Forms: cond

- The keyword cond enables an algorithm to branch in an arbitrary number of directions. It usually appears in the form

\[
\text{(cond (test-1 expr-1)
     (test-2 expr-2)
     \ldots
     (test-n expr-n)
     \text{(else else-expr))}
\]

The tests (test-1, test-2, \ldots) are performed in sequence until one of them evaluates to something other than #f. Then the expr that immediately follows that test is evaluated and returned. In the event that all tests return #f, then, the else-expr is evaluated and returned.

\[
\text{(define (silly n)
     (cond ((even? n) (+ n 2))
           ((> n 13) (* n n))
           (else (silly (* 2 n)))))}
\]
Special Forms: quote

The keyword `quote` suppresses the evaluation of its argument. For example, if \( x \) is still bound to the numerical value 4, then

\[
(quote \ x) \Rightarrow x,
\]

whereas,

\[
x \Rightarrow 4.
\]

`quote` is so useful that it is abbreviated using the apostrophe. Thus \( \textquotesingle x \) is equivalent to \((quote \ x)\). With `quote` it is possible to bind a symbols to other symbols or other literal expressions:

\[
\text{(define a \textquotesingle b)}
\]
\[
a \Rightarrow b
\]

\[
\text{(define c \textquotesingle (x y (z)))}
\]
\[
c \Rightarrow (x y (z))
\]
Functions that dissect lists: \texttt{car} and \texttt{cdr}

- \texttt{car} returns the first element of a list. For example
  \[(\texttt{car } '(\texttt{apple pear plum}))\] returns \texttt{apple}.

- \texttt{cdr} ("could-er") returns what is left in a list after the first element is removed:
  \[(\texttt{cdr } '(\texttt{apple pear plum}))\] returns \texttt{(pear plum)}.

- Generalizations:
  \[
  \begin{align*}
  (\texttt{cadr x}) & \equiv (\texttt{car (cdr x)}) \\
  (\texttt{cddr x}) & \equiv (\texttt{cdr (cdr x)}) \\
  (\texttt{caddr x}) & \equiv (\texttt{car (cdr (cdr x))})
  \end{align*}
  \]

Evaluating either \texttt{(car ’())} or \texttt{(cdr ’())} results in an \texttt{error}.

Why \texttt{car} and \texttt{cdr}? On the ancient IBM 704, on which LISP was developed, \texttt{car} stood for \textit{contents of the address part of the register}; and \texttt{cdr}, for \textit{contents of the decrement part of the register}. 
Using `list` to create a list

The function `list` is a quick way to build a list.

```
(list 'a 'b 'c) ⇒ (a b c)
(list) ⇒ ()
```
Using \texttt{cons} to construct lists

The function \texttt{cons} constructs lists and pairs:

\[
\begin{align*}
(\text{cons } '\text{plum } ()') & \Rightarrow (\text{plum}) \\
(\text{cons } '(a \ b) ' (c)) & \Rightarrow ((a \ b) \ c) \\
(\text{cons } 'a (\text{cons } 'b (\text{cons } 'c ' ()'))) & \Rightarrow (a \ b \ c) \\
(\text{cons } 'a ' b) & \Rightarrow (a . b)
\end{align*}
\]

Also note that if we define \texttt{(define \textit{lst} '(a b c))}, then since

\[
\begin{align*}
(\text{car \textit{lst}}) & \Rightarrow a, \ and, \\
(\text{cdr \textit{lst}}) & \Rightarrow (b \ c), \ so \\
(\text{cons (car \textit{lst}) (cdr \textit{lst})}) & \Rightarrow (a \ b \ c)
\end{align*}
\]
Using cons to roll dice

The built-in function `random` takes either zero or one arguments: `(random)` with no arguments, returns a random decimal number $x$, such that $0 \leq x < 1$. When called with an argument $n$ that is a positive integer, then `(random n)` returns a random integer $k$ such that $0 \leq k < n$. We can thus simulate the roll of a six-sided die with

```
(define (die)
  (+ 1 (random 6)))
```

And the roll of $n$ dice with the function

```
(define (dice n)
  (if (zero? n)
      ()
      (cons (die) (dice (- n 1)))))
```

Then the expression `(dice)` returns the empty list `()`, `(dice 1)` returns a random roll of a single die in a list, e.g., `(1)`, `(2)`, . . . , or `(6)`, and `(dice 3)` might return `(1 5 3)`, or perhaps `(6 6 6)`. 
One way to use `cons`

`cons` can be used to clone (e.g., create a copy) of a list:

```
(define (clone s) ; s is a list of atoms
  (cond ((null? s) ())
        (else (cons (car s) (clone (cdr s))))))
```

Thus,

```
(clone '(a b a c a d)) ⇒ (a b a c a d)
```

In the above `car` and `cdr` break the input list apart, and `cons` puts the pieces back together again.

`clone` acts as an identity function: it returns an exact copy of its input.

How would you extend `clone` to work with atoms as well as lists of lists? (Think about this before reading further.)
Variations of clone

Once you know how clone works, it can be modified to perform some non-trivial functions. What does the following variation do?

```
(define (anteater s)
  (cond ((null? s) '())
        ((eq? (car s) 'ant) (anteater (cdr s)))
        (else (cons (car s) (anteater (cdr s))))))
```

(The differences between anteater and clone are highlighted in red.)

(anteater '(ant bee fly ant gnat wasp ant)) returns (bee fly gnat wasp).

What does this variation do?

```
(define (alchemy s)
  (cond ((null? s) ())
        ((eq? (car s) 'lead) (cons 'gold (alchemy (cdr s))))
        (else (cons (car s) (alchemy (cdr s)))))
)
```

Please evaluate,

```
(alchemy '(tin silver lead copper zinc lead nickel tin))
```

to obtain,

```
(tin silver gold copper zinc gold nickel tin))
```
Finding the member of a set

We can use a list to represent a set. For example, the set of suits might be defined as

```
(define suits '(clubs diamonds hearts spades))
```

We can use the built-in function `memq` to determine if `hearts` is a suit:

```
(memq 'hearts suits)
```

returns

```
'(hearts spades)
```

even though

```
(memq 'stars suits))
```

returns `#f`. 
An implementation of \texttt{memq}

\texttt{memq} can be defined as

\begin{verbatim}
(define (memq x s)
  (cond ((null? s) #f)
        ((eq? (car s) x) s)
        (else (memq x (cdr s)))))
\end{verbatim}

Isn’t this another variation of \texttt{clone}?
Repeating an expression \( n \) times

How can we define a function called \( \text{repeat} \), so that

\[
\begin{align*}
(\text{repeat } 'a 7) & \Rightarrow (a \ a \ a \ a \ a \ a \ a), \\
(\text{repeat } 'a 0) & \Rightarrow ()\
\end{align*}
\]

Answer:

\[
\text{(define (repeat s n)}
\begin{align*}
(\text{if } (= \ n \ 0) & \\
(()) & \\
(\text{cons s (repeat s (- n 1))}))
\end{align*}
\text{)}
\]

Note the similarity in form of the above to \text{factorial}, and other functions that we defined earlier.
Reversing the order of a list?

How can we copy a list so that it appears in reverse order, so that

\[(\text{reverse1 } '(a b c d)) \Rightarrow (d c b a)\]
\[(\text{reverse1 } '()) \Rightarrow ()\]

It is helpful to first create an auxiliary function, which we shall call \text{reverse2} which does something similar using two arguments, \text{in} and \text{out}.

\[
\text{(define (reverse2 in out)}
\text{(if (null? in)}
\text{out}
\text{(reverse2 (cdr in) (cons (car in) out))}))
\]

\[
\text{(define (reverse1 in)}
\text{(reverse2 in '()))}
\]

Racket actually has a built in function called \text{reverse} that behaves just like our \text{reverse1}, but now we know how to build it ourselves. Note that all we really used were \text{car}, \text{cdr}, and \text{cons}.
Special Forms: and

The special form `and` accepts an arbitrary number of arguments. If no arguments are provided, then `and` returns `#t`. It returns `#f` if it has an argument that evaluates to `#f`. Otherwise the value of the last argument is returned. Thus for example

\[
\begin{align*}
(\text{and} \ #t \ #t) & \Rightarrow \ #t \\
(\text{and} \ #t \ #f) & \Rightarrow \ #f \\
(\text{and} \ #f \ #t) & \Rightarrow \ #f \\
(\text{and} \ #f \ #f) & \Rightarrow \ #f
\end{align*}
\]

However, `and` accepts an arbitrary number of arguments, so for example,

\[
\begin{align*}
(\text{and}) & \Rightarrow \ #t & \text{None of the arguments are } #f \\
(\text{and} \ #f) & \Rightarrow \ #f & \text{The only argument is } #f \\
(\text{and} \ '(a \ b \ c) \ #\backslash f \ 3) & \Rightarrow \ 3 & \text{None of its arguments are } #f: \\
& & \text{the second represents the character ‘} f’
\end{align*}
\]
Special Forms: and (cont.)

The procedure and qualifies as a special form because it is oftentimes not necessary to evaluate every argument: argument evaluation stops after the first #f is encountered. The behavior of and is revealed by including a few display expressions. Note that the procedure (display expr) evaluates its argument, prints its value in the interaction window, but then returns nothing. Usually we use display to print messages in the interaction window, as in (display "hello!") which prints "hello!". Evaluating,

\[
\text{(and (display 1) (display 2) #t (display 3) #t)}
\]

prints 123 and returns the value #t, But

\[
\text{(and (display 1) (display 2) #f (display 3) #t)}
\]

prints only 12 and returns #f. Thus, (display 3) was not evaluated.
Special Forms: or

The special form `or` accepts an arbitrary number of arguments. If no arguments are provided, then `or` returns `#f`. It returns `#f` only if every argument evaluates to `#f`. Otherwise the first argument that does not evaluate to `#f` is returned. Thus for example

\[
\begin{align*}
(\text{or } #t & \ #t) \Rightarrow #t \\
(\text{or } #t & \ #f) \Rightarrow #t \\
(\text{or } #f & \ #t) \Rightarrow #t \\
(\text{or } #f & \ #f) \Rightarrow #f
\end{align*}
\]

However, `or` accepts an arbitrary number of arguments, so for example,

\[
\begin{align*}
(\text{or}) & \Rightarrow #f \\
(\text{or } #t) & \Rightarrow #t \\
(\text{or } #f \ 3 \ (\text{display } 1) \ #t) & \Rightarrow 3 \quad \text{The first argument that is not } #f \\
& \text{evaluates to } 3; \ (\text{display } 1) \text{ is not evaluated.}
\end{align*}
\]
Back to solving mazes?

Given that we have defined the Hampton Court maze as

```
(define *maze* ’((a (b)) (b (a c d)) (c (b)) (d (b e f))
  (e (d)) (f (d g h)) (g (f)) (h (f i j))
  (i (h j l)) (j (h i k)) (k (j)) (l (i m n))
  (m (l)) (n (l o p)) (o (n)) (p (n)))))
```

what does the following variation of clone do?

```
(define (neighbors node graph)
  (cond ((null? graph) ’())
    ((eq? node (caar graph)) (cadar graph))
    (else (neighbors node (cdr graph)))))
```

Evaluate (neighbors ’j *maze*).
Computing the length of a list

The length of a list is defined as the number of elements it contains. Thus, the length of \((h \ i \ k)\) is 3.

The built-in function \texttt{length} performs this task. Thus,

\[
\text{(length '}(h \ i \ k)) \Rightarrow 3.
\]

If \texttt{length} were not built-it, it could be implemented like this:

\[
\text{(define (length s)}
\begin{align*}
(\text{cond} & \ (\text{null? s} \ 0) \\
(\text{else} & \ (+ \ 1 \ (\text{length (cdr s))})))
\end{align*}
\]
Function composition

We can now combine `neighbors` and `length` to obtain a function `count-neighbors` that returns the number of neighbors a node in the graph:

```
(define (count-neighbors node graph)
  (length (neighbors node graph)))
```

So

```
(count-neighbors 'j *maze*) → 3.
```
How to select an arbitrary item in a list: \texttt{list-ref}

\texttt{(list-ref s n)} returns the element of list \texttt{s} with index \texttt{n}. (N.B., the first element of a list has index \texttt{0}; the last has index \texttt{L} – \texttt{1}, where \texttt{L} is the length of the list.)

\texttt{(list-ref '(apple pear plum fig) 2) ⇒ plum.}

How could one implement \texttt{list-ref}?

\begin{verbatim}
(define (list-ref s index)
  (cond ((null? s) (error "list-ref: index too large!"))
        ((= 0 index) (car s))
        (else (list-ref (cdr s) (- index 1))))
\end{verbatim}
Random walk in a maze

(define (get-random-neighbor v graph)
  (list-ref (neighbors v graph)
    (random (count-neighbors v graph)))))

(define (randomsearch start goal graph)
  (display start)
  (cond ((eq? start goal) '())
        (else (randomsearch
               (get-random-neighbor start graph) goal graph)))))
Extending clone to nested lists

(define (clone s) ; s is an s-expression (list or atom)
 (cond ((null? s) '())
       ((not (list? s)) s)
       (else (cons (clone (car s)) (clone (cdr s)))))))

More information about using cons is available in the file clone.scm available on the CS 32 web page.
append

append appends two or more lists together:

\[
\begin{align*}
(append \ (a \ b) \ (b \ c) \ (c \ d)) & \Rightarrow (a \ b \ b \ c \ c \ d) \\
(append \ 'plum \ ()) & \Rightarrow Error! \\
(append \ '(a \ b) \ (c) \ '((d))) & \Rightarrow (a \ b \ c \ (d)) \\
(append \ '(a) \ 'b) & \Rightarrow (a \ . \ b)
\end{align*}
\]
Association lists

Recall the dotted pair \((a \ . \ x)\) is created by evaluating \((\text{cons } a \ 'x)\). Consequently,

\[
\begin{align*}
\text{(car '(a \ . \ x)) } & \Rightarrow a \\
\text{(cdr '(a \ . \ x)) } & \Rightarrow x
\end{align*}
\]

A list of dotted pairs,

\[((a \ . \ x) \ (b \ . \ y) \ (c \ . \ z) \ldots \ )\]

is called an **association list**. It is a “poor man’s database,” but often handy. The built in function `assoc` can be used to retrieve records. Thus, if the symbol `a-list` is bound to the above list, then

\[
\text{(assoc 'c a-list)} \Rightarrow (c \ . \ z)
\]

Note that

\[
\text{(cdr (assoc 'c a-list)) } \Rightarrow z
\]
Local variables and \texttt{let}

On many occasions it is handy to define local variables to simplify the evaluation process, and create code that is easier to read. The keyword \texttt{let} facilitates this. The syntax

\begin{verbatim}
(let ([var-1 expr-1]
     [var-2 expr-2]
     ...
     [var-n expr-n])
  body)
\end{verbatim}

creates a “block” of code that restricts the scope of all variable defined within its first argument.

Example:

\begin{verbatim}
(let ([x (+ 2 5)] ; Creates a temporary variable x with value 7
     [y (* 2 5)]) ; Creates a temporary variable y with value 10
  (+ x y)) ; the body returns 17
            ; After the last ), x and y are no longer defined.
\end{verbatim}
Local functions and letrec

letrec acts like let, except that it can be used to define recursive functions. For example,

```
(define (factorial n)
  (letrec ((aux (lambda (n)
                  (cond ((= 0 n) 1)
                        (else (* n (aux (- n 1)))))))
           (aux n)))
```
Implementing Trémaux

Recall that each node is represented by a unique symbol, and the topology of the graph by a list of lists. Let \( p \) and \( q \) denote two arbitrary nodes that are connected by a direct path \( pq \). How can we mark the left end of the path \( pq \) with an \( N \) and the right end with an \( X \)?

\[
\begin{array}{c}
p \quad N \quad X \\
\end{array}
\]

\[
\begin{array}{c}
p \quad (p \cdot q) \quad (q \cdot p) \\
\end{array}
\]

Let \( (p \cdot q) \) denote the end of the path at node \( p \) that leads to node \( q \); and \( (q \cdot p) \) the other end of the path adjacent to node \( q \). Then the nested pairs

\[
((p \cdot q) \cdot N) \quad \text{and} \quad ((q \cdot p) \cdot X)
\]

indicate the desired information as elements of an association list. Here the path-ends \( (p \cdot q) \) and \( (q \cdot p) \) serve as keys, and \( N \) and \( X \) as the corresponding values. Note that if a particular path-end is absent from the list, we assume that no label is present.

Other useful schemes can be found.

As we generate new \( N \)s and \( X \)es, we will add an appropriate nested pair to list called \textit{marks}. 
Trémaux (cont.)

;; new-node? returns #t if the indicated node has not yet been visited, i.e.,
;; there is no reference to it in marks.
(define (new-node? node marks)
  (cond ((null? marks) #t)
        ((eq? node (caaar marks)) #f)
        (else (new-node? node (cdr marks))))

;; add-X inserts a new X-label for the indicated path-end into the front
;; of the indicated a-list marks.
(define (add-X exit marks) ; exit should be of the form (p . q)
  (cons (cons exit 'X) marks))

;; add-N inserts a new N-label for the indicated path-end into the front
;; of the indicated a-list marks.
(define (add-N entry marks) ; entry should be of the form (p . q)
  (cons (cons entry 'N) marks))
Trémaux (cont.)

;; get-entry returns a dotted pair in the form (a . b) that represents the
;; path from a, the indicated node, to node b, along a path that has not yet
;; been used in the search. If no such path exists, then get-entry returns #f.
(define (get-entry node neighbors marks)
  (if (null? neighbors)
      #f
      (let ((candidate-path (cons node (car neighbors))))
        (if (not (assoc candidate-path marks))
            candidate-path
            (get-entry node (cdr neighbors) marks)))))

;; get-exit returns the path at the indicated old node that is labeled
;; with an ’X. A dotted pair of the form (a . b) is returned, where a is the
;; indicated node, and b indicates the destination of the path. If no exit
;; path exists then #f is returned.
(define (get-exit node neighbors marks)
  (if (null? neighbors)
      #f
      (let* ((path (cons node (car neighbors)))
              (label (assoc path marks))) ; label can be #f or ((a . b) . X)
        (if (and label (eq? (cdr label) ’X))
            path
            (get-exit node (cdr neighbors) marks)))))
# A Recursive Implementation of Trémaux’s Algorithm

<table>
<thead>
<tr>
<th>Direction</th>
<th>Destination Node</th>
<th>Subsequent Action</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Forward</strong></td>
<td>New Junction (\textit{No} labeled paths.)</td>
<td>Place $X$ at exit. Select new path. Place $N$ at new entrance. March \textit{forward}.</td>
</tr>
<tr>
<td><strong>Forward</strong></td>
<td>Old Junction (\textit{Some} labeled paths.)</td>
<td>Place $N$ at exit. Turn around. March \textit{backward}.</td>
</tr>
<tr>
<td><strong>Forward</strong></td>
<td>Dead End</td>
<td>Turn around. March \textit{backward}.</td>
</tr>
<tr>
<td><strong>Forward</strong></td>
<td>Goal</td>
<td>Eureka!</td>
</tr>
<tr>
<td><strong>Backward</strong></td>
<td>Original Entrance</td>
<td>Give up!</td>
</tr>
<tr>
<td><strong>Backward</strong></td>
<td>Old Junction with \textit{some} unlabeled paths</td>
<td>Select new (unlabeled) path. Place an $N$ at new entrance. March \textit{forward}.</td>
</tr>
<tr>
<td><strong>Backward</strong></td>
<td>Old Junction with \textit{no} unlabeled paths</td>
<td>Select path labeled with $X$. March \textit{backward}.</td>
</tr>
</tbody>
</table>
tremaux: the general structure

(define (tremaux start goal graph)
  (letrec ([start? (lambda (v) (eq? v start))]
          [goal? (lambda (v) (eq? v goal))]
          [dead-end? (lambda (v neighbors)
                       (and (not (start? v))
                            (not (goal? v))
                            (= (length neighbors) 1)))]
          [forward (lambda (path marks) ... )]
          [backward (lambda (path marks) ... )])
    (forward (list start) '())))

tremaux: the forward function

(define (tremaux start goal graph)
  (letrec ( ...
    [forward (lambda (path marks)
      (let* ([node (car path)]
        [neighbors (get-neighbors node graph)]
        [last (if (null? (cdr path)) start (cadr path))]
        [exit (cons node last)])
      (cond ((goal? node)
              (tremaux-output start goal graph path
                             (add-X exit marks)))
           ((dead-end? node neighbors)
              (backward (cons last path) marks))
           ((new-node? node marks)
              (let* ([marks (if (start? node)
                            marks
                            (add-X exit marks))]
                     [entry (get-entry node neighbors marks)])
                (forward (cons (cdr entry) path)
                         (add-N entry marks)))
           (else (backward (cons (cadr path) path)
                          (add-N exit marks)))))))
    (forward (list start) '()))))
(define (tremaux start goal graph)
 (letrec ( ...
      [backward (lambda (path marks)
         (let* ([node (car path)]
            [neighbors (get-neighbors node graph)]
            [entry (get-entry node neighbors marks)])
             (cond (entry (forward (cons (cdr entry) path)
               (add-N entry marks)))
               ((start? node)
                (display "Search Failed") (newline) #f)
               (else (let ([exit (get-exit node neighbors marks)])
                (backward (cons (cdr exit) path)
               marks))))))

 (forward (list start) '()))))
Other length functions

- **vector-length** returns the number of elements in a vector
  
  \[(\text{vector-length } '(a b 32)) \Rightarrow 3.\]

- **string-length** returns the number of characters in a string
  
  \[(\text{string-length } "Hello world!") \Rightarrow 12.\]
Other Standard Functions

- `(list-ref lst n)` returns the element of list `lst` with index `n`. (N.B., the first element of a list has index 0; the last has index $L - 1$, where $L$ is the length of the list.)

  \[
  \text{(list-ref '(apple pear plum fig) 2)} \Rightarrow \text{plum.}
  \]

- `(vector-ref vec n)` returns the element of vector `vec` with index `n`. (N.B., the first element of a vector has index 0; the last has index $L - 1$, where $L$ is the length of the vector.)

  \[
  \text{(vector-ref '#(3.6 1.2 -16. 4) 2)} \Rightarrow -16.
  \]

- `(string-ref str n)` returns the character in string `str` with index `n`. (N.B., again, the first element of a string has index 0; the last has index $L - 1$, where $L$ is the length of the string.)

  \[
  \text{(string-ref "Racket is fun!" 2)} \Rightarrow \text{"\c}
  \]
More racket functions

- `(load filename)` reads and evaluates the racket expressions contained in file indicated by the string `filename`.

  `(load "abracadabra.scm") ⇒ ?`
apply

apply takes two arguments: a function, and a list. It returns the value that is returned by applying the function to the values in the list. Thus,

\[(\text{apply } \text{fn} (\text{arg1 arg2 ... argn})\]

evaluates to result of

\[(\text{fn arg1 arg2 ... argn})\]

For example,

\[(\text{apply } + (1 2 3 4 5))\]

evaluates to 15.
The apply function (cont.)

(apply + '(1 2 3 4)) ⇒ 10
(apply max '(1 2 3 4)) ⇒ 4
(apply list '(1 2 3 4)) ⇒ (1 2 3 4)

Actually the syntax of apply is more generally of the form

(apply procedure object ... list),

the arguments of procedure are obtained by concatenating the objects and list elements in the order that they appear. Only one list is allowed. Thus,

(apply + 100 200 '(1 2 3 4)) ⇒ 310
map

map applies a function of $n$ arguments to $n$ lists that have the same length:

- (map zero? '(0 0.3 -2 0)) \Rightarrow (#t #f #f #t)
- (map abs '(0 0.3 -2 0)) \Rightarrow (0 0.3 2 0)
- (map (lambda (x) (+ x 2)) '(0 0.3 -2 0)) \Rightarrow (2 2.3 0 2)
- (map (lambda (x y) (+ x y)) '(0 1 2) '(3 3 0)) \Rightarrow (3 4 2)
- (map (lambda (x y) (+ x y)) '(0 1 2) '(3 3)) \Rightarrow Error!
Question

What does the following function do?

(define (mystery x)
  (map caddr x))

(mystery '((a b c) (1 2 3) (x y z)))
Functions that return functions

Consider the following racket definition

```
(define compose
  (lambda (f g) ; The result of the 1st lambda exp. is bound to compose
    (lambda (x) ; compose returns the 2nd lambda expression
      (f (g x))))))
```

where \( f \) and \( g \) are assumed to be functions that accept a single argument. What happens if we evaluate

```
(define mys (compose car cdr))

(mys '(a b c))
```