Problems & Solutions
(and “use it or lose it”)

Racket miscellanea: let*

Welcome to DrRacket, version 6.2.1 [3m].
Language: racket [custom]; memory limit: 128 MB.
> (let ([x 10]
   [y (* x 2)])
  (+ x y))

x: undefined;
cannot reference an identifier before its definition
> (let* ([x 10]
   [y (* x 2)])
  (+ x y))
30

docs.racket-lang.org/reference/let.html
Welcome to DrRacket, version 5.3.6 [3m].
Language: racket; memory limit: 128 MB.
> ella ; an undefined variable
 ella: undefined;
cannot reference an identifier before its definition
> 'ella ; a symbol!
'ella
> '(ella 1917) ; a symbol in a list
'(ella 1917)
> (symbol? (first '(ella 1917))) ; just to prove I wasn't lying
#t
> (symbol? (second '(ella 1917))) ; an int is not a symbol
#f
> (define ella 1917) ; defining a variable
 ella ; using a variable
1917
> 'ella ; still a symbol!
'ella
> '(ella 1917) ; STILL a symbol in a list!
'(ella 1917)
> (list ella 1917) ; it's a subtle difference
'(1917 1917)
Racket miscellanea: range & sort

Welcome to DrRacket, version 5.3.6 [3m].
Language: racket; memory limit: 128 MB.
> (define values (range 10))
> values
'(0 1 2 3 4 5 6 7 8 9)
> (sort values <)
'(0 1 2 3 4 5 6 7 8 9)
> (sort values >)
'(9 8 7 6 5 4 3 2 1 0)

docs.racket-lang.org/reference/pairs.html
Racket miscellanea: range & sort

```racket
(define jazz-stars '( (Armstrong 1901) (Brubeck 1920) (Coleman 1930) (Ellington 1899) (Fitzgerald 1917) (Holiday 1915) (Monk 1917) ))

(sort jazz-stars #:key second <)
'((Ellington 1899)
  (Armstrong 1901)
  (Holiday 1915)
  (Fitzgerald 1917)
  (Monk 1917)
  (Brubeck 1920)
  (Coleman 1930))

(map reverse (sort jazz-stars #:key second <))
'((1899 Ellington)
  (1901 Armstrong)
  (1915 Holiday)
  (1917 Fitzgerald)
  (1917 Monk)
  (1920 Brubeck)
  (1930 Coleman))
```

docs.racket-lang.org/reference/pairs.html
Racket miscellanea: member

Welcome to DrRacket, version 5.3.6 [3m].
Language: racket; memory limit: 128 MB.

> (range 10)
'(0 1 2 3 4 5 6 7 8 9)
> (member 10 (range 10))
#f
> (member 0 (range 10))
'(0 1 2 3 4 5 6 7 8 9)
> (member 5 (range 10))
'(5 6 7 8 9)

docs.racket-lang.org/reference/pairs.html
Welcome to DrRacket, version 6.2.1 [3m].
Language: racket [custom]; memory limit: 128 MB.
>(append (range 10) (range 11 20))
'(0 1 2 3 4 5 6 7 8 9 11 12 13 14 15 16 17 18 19)
>(append (range 11 20) (range 10))
'(11 12 13 14 15 16 17 18 19 0 1 2 3 4 5 6 7 8 9)
>
The Seven Bridges of Königsberg

Can you:
- start at point A,
- cross every bridge only once,
- and return to point A?

Leonard Euler

No!
We need a model
Graphs!

A set of nodes/vertices (places), and a set of edges (links)
Graphs represent relationships

Like what?
A node is …
An edge is…
Undirected graphs
Directed Graphs
### A Racket interface for graphs

Data structure, manipulated via constructors, accessors, and operations

<table>
<thead>
<tr>
<th>Constructors</th>
<th>Accessors</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>(make-edge (&lt;n_s&gt; \ &lt;n_d&gt;)</em></td>
<td><em>(src &lt;edge&gt;)</em> (\text{dst} &lt;edge&gt;)*</td>
<td><em>(kids &lt;node&gt; &lt;graph&gt;)</em></td>
</tr>
<tr>
<td><em>(empty-graph)</em></td>
<td><em>(empty-graph? &lt;graph&gt;)</em></td>
<td><em>(reachable? (&lt;n_s&gt; \ &lt;n_d&gt; &lt;graph&gt;)</em></td>
</tr>
<tr>
<td><em>(add-edge &lt;edge&gt; &lt;graph&gt;)</em></td>
<td><em>(node-list &lt;graph&gt;)</em></td>
<td><em>(\text{...})</em></td>
</tr>
<tr>
<td>(\text{...})*</td>
<td><em>(edge-list &lt;graph&gt;)</em></td>
<td><em>(\text{...})</em></td>
</tr>
<tr>
<td>(\text{...})*</td>
<td><em>(remove-edge &lt;edge&gt; &lt;graph&gt;)</em></td>
<td>(\text{...})*</td>
</tr>
</tbody>
</table>
Let’s practice: kids

Hint: How to get all the edges with node as their source? How to find where they lead?

(define (kids node G)
  ; assume G is a directed graph: src → dst

(Your response)
Let’s practice: kids

Hint: How to get all the edges with node as their source? How to find where they lead?

(define (kids node G)
    (let* ([edges     (edge-list G)]
            [adjacent? (lambda (e) (equal? (src e) node))]
            [adjacent  (filter adjacent? edges)])
        (map dst adjacent)))

(Your response)
Problems & Solutions
Three problems to consider

uniq
Given a list L, create a new list L' that contains only the unique elements of L, in the order they appear in L.

sublists
Given a list L, generate a list L' of all sublists that can be made from the elements of L (elements must appear in same order).

reachable?
Given a graph G, starting point a and ending point b, determine whether b is reachable from a in G.
> (uniq '(california))
  '(california)

> (uniq '(mudd))
  '(mudd)

> (uniq '(mississippi))
  '(mississippi)
> (sublists '())
  '(())

> (sublists '(1))
  '(((1) ())

> (sublists '(1 2))
  '(((1 2) (1) (2) ())
How hard are these problems?

<table>
<thead>
<tr>
<th></th>
<th>cost metric</th>
<th>worst-case input</th>
<th>worst-case cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>uniq</td>
<td>comparisons</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sublists</td>
<td>number of sublists</td>
<td></td>
<td></td>
</tr>
<tr>
<td>reachable?</td>
<td>number of edges traversed</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## How hard are these problems?

<table>
<thead>
<tr>
<th>cost metric</th>
<th>worst-case input</th>
<th>“ideal” worst-case cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>uniq</td>
<td>comparisons</td>
<td>all are unique</td>
</tr>
<tr>
<td>sublists</td>
<td>number of sublists</td>
<td>everything is the worst (and the best!)</td>
</tr>
<tr>
<td>reachable?</td>
<td>number of edges traversed</td>
<td>b is not reachable from a</td>
</tr>
</tbody>
</table>
“Use it or lose it”
(a recursive search technique)
> (uniq '(california))
  '(california)

> (uniq '(mudd))
  '(mudd)

> (uniq '(mississippi))
  '(mississippi)
Solution technique: “use it or lose it”

a recursive search strategy

```
(define (uniq L)
  (if (empty? L)
      '()
      (let* ([it      (first L)]
              [lose-it (uniq (rest L))]
              [use-it  (cons it lose-it)])
        (if (member it lose-it)
            lose-it
            use-it)))
```

base case

split

“combine”
> (sublists '())
  '((()))

> (sublists '(1))
  '(((1) ()))

> (sublists '(1 2))
  '(((1 2) (1) (2) ())))
Solution technique: “use it or lose it”
a recursive search strategy

(define (sublists L)
  (if (empty? L)
      '(())
      (let* ([it      (first L)]
             [lose-it (sublists (rest L))]
             [use-it  (map (lambda (l) (cons it l)) lose-it)])
        (append use-it lose-it))))
Solution technique: “use it or lose it”

a recursive search strategy

(define (reachable? a b G)
  (cond 
    [(equal? a b)     true]
    [(empty-graph? G) false]
    [else (let* ([edges (edge-list G)]
                 [it    (first edges)]
                 [subG  (remove-edge it G)]
                 [lose-it (reachable? a b subG)]
                 [use-it  (and (reachable? a (src it) subG)
                                (reachable? (dst it) b subG))])
             (or use-it lose-it))])))
The cost of `uniq` measured in comparisons

```scheme
(define (uniq L)
  (if (empty? L)
      '()
      (let* ([it (first L)]
              [lose-it (uniq (rest L))])
        (if (member it lose-it)
            lose-it
            (cons it lose-it)))))
```

\[ T(0) = 0 \]
\[ T(1) = T(0) + 0 = 0 \]
\[ T(n) = T(n-1) + n-1 \]
The cost of sublists
measured in number of sublists generated

\[ T(0) = 1 \]
\[ T(1) = 2 \times T(0) = 2 \]
\[ T(n) = 2 \times T(n-1) \]

(define (sublists L)
  (if (empty? L)
      '(())
      (let* ([it     (first L)]
             [lose-it (sublists (rest L))]
             [use-it  (map (lambda (l) (cons it l)) lose-it)])
        (append use-it lose-it))))
The cost of reachable?
measured in number of edges traversed, i.e., calls to remove-edge

\[ T(0) = 0 \]
\[ T(1) = 1 + 3 \times T(0) = 1 \]
\[ T(n) = 1 + 3 \times T(n-1) \]

\[
\begin{align*}
\text{define} \ (\text{reachable}? \ a \ b \ G) \\
\ (\text{cond} \ [(\text{equal}? \ a \ b) \ \text{true}] \\
\ [(\text{empty-graph}? \ G) \ \text{false}] \\
\ [\text{else} \ (\text{let}* \ ([\text{edges} \ \text{(edge-list} \ G)])] \\
\ [\text{it} \ \text{(first} \ \text{edges})] \\
\ [\text{subG} \ \text{(remove-edge} \ \text{it} \ G)] \\
\ [\text{lose-it} \ (\text{reachable}? \ a \ b \ \text{subG})] \\
\ [\text{use-it} \ \text{(and} \ (\text{reachable}? \ a \ (\text{src} \ \text{it}) \ \text{subG})] \\
\ [\text{and} \ (\text{reachable}? \ \text{(dst} \ \text{it}) \ b \ \text{subG})])] \\
\ (\text{or} \ \text{use-it} \ \text{lose-it}))]]
\end{align*}
\]
<table>
<thead>
<tr>
<th></th>
<th>cost metric</th>
<th>worst-case input</th>
<th>“ideal” worst-case cost</th>
<th>UIOLI cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>uniq</td>
<td>comparisons</td>
<td>all are unique</td>
<td>n</td>
<td>O(n^2)</td>
</tr>
<tr>
<td>sublists</td>
<td>number of sublists</td>
<td>everything is the worst (and the best!)</td>
<td>2^n</td>
<td>O(2^n)</td>
</tr>
<tr>
<td>reachable?</td>
<td>number of edges traversed</td>
<td>b is not reachable from a</td>
<td>n</td>
<td>O(3^n)</td>
</tr>
</tbody>
</table>