Programming with Recursion and Symbolic Expressions

CIS 352 — Spring 2020
Kris Micinski
Calculating factorial in Racket

(define (factorial n)
  (if (= n 0)
      1
      (* n (factorial (sub1 n))))

Example
Calculating factorial in Racket

\[(\lambda (x) (x \ x)) (\lambda (x) (x \ x))\]

\[
(\text{define (factorial n)}
\begin{align*}
(\text{if} \ (= \ n \ 0) & \ 1 \\
(\ast \ n \ (\text{factorial} \ (\text{sub1} \ n))) & \end{align*})
\]

Defines base case
Calculating factorial in Racket

(define (factorial n)
  (if (= n 0)
      1
      (\(*\ n (factorial (sub1 n))\))))

and **inductive / recursive** case
(define (factorial n)
  (if (= n 0)
      1
      (* n (factorial (sub1 n))))

We can think of recursion as “substitution”

> (factorial 2)
We can think of recursion as “substitution”

> (factorial 2)

= (if (= 2 0)
   1
   (* 2 (factorial (sub1 2))))

Copy defn, substitute for argument n
(define (factorial n)
  (if (= n 0)
      1
      (* n (factorial (sub1 n))))

We can think of recursion as “substitution”

> (factorial 2)
= (if (= 2 0)
    1
    (* 2 (factorial (sub1 2))))
= (if #t 1 (* 2 (factorial (sub1 2))))

Evaluate if
(define (factorial n)
  (if (= n 0)
      1
      (* n (factorial (sub1 n))))

We can think of recursion as "substitution"

> (factorial 2)
= (if (= 2 0)
   1
   (* 2 (factorial (sub1 2))))
= (if #t 1 (* 2 (factorial (sub1 2))))
= (* 2 (factorial (sub1 2)))
(define (factorial n)
  (if (= n 0)
    1
    (* n (factorial (sub1 n))))

We can think of recursion as “substitution”

> (factorial 2)
= (if (= 2 0)
    1
    (* 2 (factorial (sub1 2))))
= (if #t 1 (* 2 (factorial (sub1 2))))
= (* 2 (factorial (sub1 2)))
= (* 2 (factorial 1))

Evaluate \text{sub1}
(define (factorial n)
  (if (= n 0)
      1
      (* n (factorial (sub1 n))))
)

We can think of recursion as “substitution”

> (factorial 2)
= (if (= 2 0)
    1
    (* 2 (factorial (sub1 2))))
= (if #t 1 (* 2 (factorial (sub1 2))))
= (* 2 (factorial (sub1 2)))
= (* 2 (factorial 1))
= (* 2 (if (= 1 0) 1 (* n (factorial (sub1 1)))))
(define (factorial n)
  (if (= n 0)
      1
      (* n (factorial (sub1 n)))))

= (* 2 (if (= 1 0)
      1
      (* 1 (factorial (sub1 1))))))
= (* 2 (* 1 (factorial (sub1 1))))
= (* 2 (* 1 (factorial 0)))
= (* 2 (* 1 (if (= 0 0) 1 ...)))
= (* 2 (* 1 (if #t 1 ...)))
= (* 2 (* 1 1))
= (* 2 1)
= 2
Example

\[
\text{(define \texttt{(factorial n)})}
\]
\[
\text{(if (= n 0)
1
(* n (factorial \texttt{(sub1 n)})))}
\]

= (* 2 (if (= 1 0)
1
(* 1 (factorial \texttt{(sub1 1)})))

= (* 2 (* 1 (factorial \texttt{(sub1 1)})))

= (* 2 (* 1 (factorial 0)))

= (* 2 (* 1 (if (= 0 0) 1 ...)))

= (* 2 (* 1 (if #t 1 ...)))

= (* 2 (* 1 1))

= (* 2 1)

This is “textual reduction” semantics

= 2

More on this later
...  
= (* 2 (if (= 2 0) 
  1 
  (* n (factorial (sub1 2)))))
= (* 2 (factorial 1))
= ...
= (* 2 (* 1 1))
= (* 2 1)
= 2

Notice we’re building a big stack of calls to *

Then recursion “bottoms out:”
returns back to finish the work

(More on this next week...)}
(define (log2 n)
  (if (= n 1) 0 (+ 1 (log2 (/ n 2))))
)

(log2 2)
= (if (= 2 1) 0 (+ 1 (log2 (/ 2 2))))
= ???
= ...
= ???
Exercise

(define (log2 n)
  (if (= n 1) 0 (+ 1 (log2 (/ n 2)))))

(log2 2)
= (if (= 2 1) 0 (+ 1 (log2 (/ 2 2))))
= (+ 1 (log2 (/ 2 2)))
= (+ 1 (log2 1))
= (+ 1 (if (= 1 1) 0 (+ 1 (log2 (/ 1 2))...))
= (+ 1 (if #t 0 (+ 1 (log2 (/ 1 2))...))
= (+ 1 0)
= 1
Exercise

Write the definition of \( (\text{fib } n) \) in Racket using the following definition:

\[
\text{fib}(n) = \begin{cases} 
0 & n = 0 \\
1 & n = 1 \\
\text{fib}(n - 1) + \text{fib}(n - 2) & \text{otherwise}
\end{cases}
\]
(define (fib n)
  (if (or (= n 0) (= n 1))
      n
      (+ (fib (- n 1)) (fib (- n 2)))))
**Question:** what is the big-O time complexity of this implementation?

```
(define (fib n)
  (if (or (= n 0) (= n 1))
      n
      (+ (fib (- n 1)) (fib (- n 2))))
)```
Answer: $O(2^n)$ or exponential
(Fun fact: actually $\varphi^n$, where $\varphi$ is the golden ratio)

\[
\text{(define (fib n)} \ \\
\text{ (if (or (= n 0) (= n 1)) n)} \ \\
\text{ (+ (fib (- n 1)) (fib (- n 2)))))}
\]
We say that this algorithm uses a “top-down” approach

\[
\text{(define (fib n)} \\
\text{  (if (or (= n 0) (= n 1)) n)} \\
\text{    (+ (fib (- n 1)) (fib (- n 2)))})
\]

Because it calculates each number by first calculating the previous two fibonacci numbers
\[(\text{fib } n) \rightarrow (\text{fib } n-1) \rightarrow (\text{fib } n-2) \rightarrow (\text{fib } n-3) \rightarrow (\text{fib } n-4) \rightarrow \ldots \]

\[\text{etc...}\]
Lots of redundant work
Instead, use dynamic programming:
design a recursive solution top-down, but implement
as a bottom-up algorithm!

Start with first two, then build up
Instead, use **dynamic programming:**
design a recursive solution top-down, but implement as a bottom-up algorithm!
Key idea: only need to look at **two most recent** numbers
Accumulate via arguments

\[
\begin{align*}
&\lambda x (x \ x)
\end{align*}
\]

(\begin{align*}
&\lambda x (x \ x)
\end{align*})

Accumulate via arguments

```
(define (fib-h i n0 n1)
  (if (= i 0)
      n0
      (fib-h (- i 1) n1 (+ n0 n1))))

(define (fib n) (fib-h n 0 1))
```
(define (fib-h i n0 n1)
  (if (= i 0)
      n0
      (fib-h (- i 1) n1 (+ n0 n1)))))

(define (fib n) (fib-h n 0 1))

**Question**: what is the runtime complexity of `fib`?
Exercise

\[(\text{define (fib-h i n0 n1)} \rightarrow \text{if (} = i 0) n0 (\text{fib-h (} - i 1) n1 (+ n0 n1)))]\)

\[(\text{define (fib n)} \rightarrow (\text{fib-h n 0 1)}\)]

**Answer:** \(O(n), \text{fib-helper runs from } n \text{ to } 0\)
Consider how \texttt{fib-h} executes

\begin{verbatim}
(define (fib-h i n0 n1)
  (if (= i 0)
      n0
      (fib-h (- i 1) n1 (+ n0 n1))))

(define (fib n) (fib-h n 0 1))
\end{verbatim}
(fib-helper 3 0 1)
= (if (= 3 0) 0 (fib-h (- 3 1) 1 (+ 0 1)))
= ... 
= (fib-h 2 1 1)
= (if (= 2 0) 1 (fib-h (- 2 1) 1 (+ 1 1)))
= ...  
= (fib-h 1 1 2)

Notice that we don’t get the “stacking” behavior: recursive calls don’t grow the stack
This is because `fib-h` is tail recursive

\[
\text{(define (fib-h i n0 n1)}
\begin{align*}
\text{  (if (= i 0)} & \n0 \\
\text{    n0} & \n0 \\
\text{        (fib-h (- i 1) n1 (+ n0 n1)))}
\end{align*}
\]

\[
\text{(define (fib n) (fib-h n 0 1))}
\]

Intuitively: a callsite is in tail-position if it is the \textbf{last thing} a function will do before exiting

(We call these \textbf{tail calls})
This is because `fib-h` is **tail recursive**

Both of these are tail calls

\[
\text{(define (fib-h \(i\) \(n0\) \(n1\))}
\text{(\(if\) (= \(i\) 0) \(n0\)
\text{(fib-h \((-\ i\) 1) \(n1\) (+ \(n0\) \(n1\))))}
\text{)}
\text{)}
\]

\[
\text{(define (fib \(n\) (fib-h \(n\) 0 1))}
\text{)}
\]

Intuitively: a callsite is in **tail-position** if it is the **last thing** a function will do before exiting

(We call these **tail calls**)
Tail calls / tail recursion

• Unlike calls in general, **tail calls** do not affect the stack:
  • Tail calls *do not grow* (or shrink) the stack.
  • They are more like a goto/jump than a normal call.

• A subexpression is in **tail position** if it’s the last subexpression to run, whose return value is also the value for its parent expression:
  • In `(let ([x rhs]) body); body is in tail position…`
  • In `(if grd thn els); thn & els are in tail position…`

• A function is **tail recursive** if all recursive calls in tail position

• Tail-recursive functions are analogous to loops in imperative langs
Exercise

Which of the following is tail recursive?

(define (length-0 l)
  (if (null? l)
      0
      (+ 1 (length-0 (cdr l))))

(define (length-1 l n)
  (if (null? l)
      n
      (length-1 (cdr l) (+ n 1))))
Exercise

Answer

(define (length-0 l)
  (if (null? l)
      0
      (+ 1 (length-0 (cdr l)))))

(define (length-1 l n)
  (if (null? l)
      n
      (length-1 (cdr l) (+ n 1))))

Not tail recursive
Adds (+ 1 _) operation to stack

Is tail recursive!
Call to length-1 in tail position
Structured Data

• A list is an example of a recursive data structure
  • Defined via a base case and inductive case:
    • A list is either the \textbf{empty list} / \textbf{null} / ‘()
    • Or a \textbf{cons cell} of any element and \textbf{another list}
  • We can check whether it’s \textbf{null?} or \textbf{cons?} or \textbf{list?}
  • Can access via \textbf{car} and \textbf{cdr}; or \textbf{first} and \textbf{rest}
    • Many recursive functions on lists built using these
Exercise

Write a function to calculate the sum of a list

; (sum-list '(1 2)) is 3
(define (sum-list l)
  ...)

38
Write a function to calculate the sum of a list

; (sum-list '(1 2)) is 3
(define (sum-list l)
  ...)

Answer (one of many)

(define (sum-list l)
  (if (eq? l '())
      0
      (+ (car l)
          (sum-list (cdr l)))))
Accumulator Passing

• Many functions can be written by passing an accumulator: a value that is repeatedly extended to obtain a final value.

• Esp. in tail-recursive / looping algorithms; e.g.:

```
(define (sum-list l)
  (define (sum-loop l acc)
    (if (empty? l)
        acc
        (sum-loop (rest l)
                  (+ acc (first l))))
  (sum-loop l 0))
```
S-exprs  (*symbolic expressions*)

• The **S-expression** is our parenthesized notation for a list
  • Can use lists to group data common to some structure
• We can **tag** expressions with a symbol to note its “type”
  • ‘(point 2 3)
  • ‘(square (point 0 1) 5)
• Can define “constructor” functions
  
  (define (mk-point x y)  
    (list ‘point x y))

  (define (mk-square pt0 len)  
    (list ‘square pt0 len))
quasi-quotes

- Racket offers **quasi-quotes** to build S-expressions fast

- `(,x y 3)` is equivalent to `(list x `y `3)`

  - I.e., Racket splices in values that are unquoted via `,`
  
  - `(quasiquote ...)` will substitute any expression ,e with the return value of e within the quoted S-expression

- Works multiple levels deep:

  - `(square (point ,x0 ,y0) (point ,x1 ,y1))`

- Can unquote entire expressions:

  - `(point ,(+ 1 x0) ,(− 1 y0))`
Define mk-point and mk-square using Quasi-quotation:

(define (mk-point x y)
  (list ‘point x y))

(define (mk-square pt0 pt1)
  (list ‘square pt0 pt1))
Define mk-point and mk-square using Quasi-quotation:

```
(define (mk-point x y)
  (list ‘point x y))

(define (mk-square pt0 pt1)
  (list ‘square pt0 pt1))
```

**Answer**

```
(define (mk-point x y)
  `(point ,x ,y))

(define (mk-square pt0 pt1)
  `(square ,pt0 ,pt1))
```
Pattern Matching

- Racket also has **pattern matching**
  - \((\text{match } e \left[\text{pat}_0 \ \text{body}_0\right] \left[\text{pat}_1 \ \text{body}_1\right] \ldots)\)
- Evaluates \(e\) and then checks each **pattern**, in order
- Pattern can bind variables, body can use pattern variables
- Many patterns (check docs to learn various useful forms)
- Patterns checked in order, first matching body is executed
  - Later bodies won’t be executed, even if they also match!
- E.g., (match `(1 2 3)
  `(`(`a b) b)`
  `(`(`a . b) b)`
  ); returns `(2 3)"
Matching a literal

\[
\text{(match e e)
\begin{array}{ll}
\text{[‘hello ‘goodbye]} & \\
\text{[\(? \text{number? } n \)? (+ n 1)]} & \\
\text{[\(? \text{nonnegative-integer? } n \)? (+ n 2)]} & \\
\text{[(cons x y) x]} & \\
\text{[\`(,a0 ,a1 ,a2) (+ a1 a2)]} &
\end{array}
\]
\]
Matches when $e$ evaluates to some number?

(match $e$

[‘hello ‘goodbye]

[[(? number? n) (+ n 1)]

[[(? nonnegative-integer? n) (+ n 2)]

[(cons x y) x]

[`(,a0 ,a1 ,a2) (+ a1 a2)]

(binds n)
(match e
[‘hello ‘goodbye]
[?(number? n) (+ n 1)]
[?(nonnegative-integer? n)
(+ n 2)]
[cons x y) x]
[`(,a0 ,a1 ,a2) (+ a1 a2)])

Never matches!
Subsumed by previous case!
(match e
['hello 'goodbye]
[(? number? n) (+ n 1)]
[(? nonnegative-integer? n) (+ n 2)]
[(cons x y) x]
[,(a0 ,a1 ,a2) (+ a1 a2)]
)

Matches a cons cell, binds x and y
(match e
    ['hello 'goodbye]
    [(? number? n) (+ n 1)]
    [(? nonnegative-integer? n)
      (+ n 2)]
    [(cons x y) x]
    [`((a0 ,a1 ,a2) (+ a1 a2))] )

Matches a list of length three
Binds first element as a0, second as a1, etc…
Called a “quasi-pattern”

Can also test predicates on bound vars:

`((,(? nonnegative-integer? x) ,(? positive? y))"
(match e
   ['hello 'goodbye]
   [(? number? n) (+ n 1)]
   [(? nonnegative-integer? n) (+ n 2)]
   [(cons x y) x]
   [\(,a0 ,a1 ,a2\) (+ a1 a2)]
   [_ 23])

Can also have a default case
Exercise

Define a function \texttt{foo} that returns:
- twice its argument, if its argument is a number?
- the first two elements of a list, if its argument is a list of length three, as a list
- the string “error” if it is anything else

\begin{verbatim}
(define (foo x)
  (match x
    [(? ...) ...]
    ...))
\end{verbatim}
Exercise

Define a function `foo` that returns:
- twice its argument, if its argument is a number?
- the first two elements of a list, if its argument is a list of length three, as a list
- the string “error” if it is anything else

**Answer (one of many)**

```scheme
(define (foo x)
  (match x
    [(? number? n) (* n 2)]
    [`(,a ,b ,_) `(,a ,b)]
    [_ "error"]))
```
Define a function `foo` that returns:
- twice its argument, if its argument is a number?
- the first two elements of a list, if its argument is a list of length three, as a list
- the string “error” if it is anything else

**Answer (one of many)**

```scheme
(define (foo x)
  (match x
    [(? number? n) (* n 2)]
    [`(,a ,b ,_) `(,a ,b)]
    [_ "error"]))
```

Observe how quasipatterns and quasiquotes interact.
Structural Recursion

• Structural recursion
  • Recurs on some smaller piece of the input obtained by destructing (e.g., matching) on it.

• Easy to prove termination
  • Code is making input smaller at each recursive step, thus will eventually bottom out

• Much of the code you will write is structurally recursive

• But some things cannot be expressed in a structurally recursive way
  • E.g., generative recursion, other algorithms, …
Consider that we define trees as follows:

```
(define (tree? t)
  (match t
    [`(leaf ,n) #t]
    [`(node ,(? tree? t0) ,(? tree? t1)) #t]
    [_ #f]))
```

Assuming trees are sorted, write a recursive function using match patterns, (least t) to get the smallest element in the tree (i.e., bottom left leaf).

(least (node (leaf 0) (leaf 1)) should be 0
(Hint: look at the definition of tree?)
Generative Recursion

• Generative recursion
  • Recurs on some structure built / calculated from input
• Not as easy (in general) to prove termination
  • How do we know it won’t just loop forever?
• Strictly more powerful than structural recursion
  • Some programs we can’t write w/ just structural recursion
  • E.g., QuickSort
QuickSort is a popular and fast sorting comparative sorting algorithm with $O(n \times \log(n))$ complexity

• To sort list $l$, first choose a pivot element (arbitrary), $p$, from $l$
• Next, construct $l'$ of the elements in $l$ that are $< p$
• Also, construct $l''$ of the elements in $l$ that are $> p$
• Now, return…
  • $\text{QuickSort}(l') + [p] + \text{QuickSort}(l'')$
**QuickSort** is a popular and fast sorting comparative sorting algorithm with $O(n \log(n))$ complexity

- To sort list $l$, first choose a **pivot** element (arbitrary), $p$, from $l$
- Next, construct $l'$ of the elements in $l$ that are $< p$
- Also, construct $l''$ of the elements in $l$ that are $> p$
- Now, return…
  - $\text{QuickSort}(l') + [p] + \text{QuickSort}(l'')$
**Example**

**QuickSort** is a popular and fast sorting comparative sorting algorithm with $O(n\log(n))$ complexity

- To sort list $l$, first choose a **pivot** element (arbitrary), $p$, from $l$
- Next, construct $l'$ of the elements in $l$ that are $< p$
- Also, construct $l''$ of the elements in $l$ that are $> p$
- Now, return…
  - $\text{QuickSort}(l') + [p] + \text{QuickSort}(l'')$

```
-5 < 0 < 1 < 3 Pivot 2 >
```
**QuickSort** is a popular and fast sorting comparative sorting algorithm with $O(n \log n)$ complexity.

- To sort list $l$, first choose a **pivot** element (arbitrary), $p$, from $l$.
- Next, construct $l'$ of the elements in $l$ that are $< p$.
- Also, construct $l''$ of the elements in $l$ that are $> p$.
- Now, return...
  - QuickSort($l'$) ++ [p] ++ QuickSort($l''$)

Now sort these!

Now sort these! Pivot >
**Example**

**QuickSort** is a popular and fast sorting comparative sorting algorithm with O(n*log(n)) complexity

- To sort list l, first choose a **pivot** element (arbitrary), p, from l
- Next, construct l’ of the elements in l that are < p
- Also, construct l” of the elements in l that are > p
- Now, return…
  - QuickSort(l’) ++ [p] ++ QuickSort(l”)

```
-5  0  1  3  2
Pivot  <  <  Pivot  >
```
**QuickSort** is a popular and fast sorting comparative sorting algorithm with O(n*log(n)) complexity

- To sort list \( l \), first choose a **pivot** element (arbitrary), \( p \), from \( l \)
- Next, construct \( l' \) of the elements in \( l \) that are < \( p \)
- Also, construct \( l'' \) of the elements in \( l \) that are > \( p \)
- Now, return…
  - \( \text{QuickSort}(l') + [p] + \text{QuickSort}(l'') \)

Now run quicksort on **these**  

Pivot >

\[
\left( \lambda x \right) \left( x \ x \ x \right)
\]

\[
\left( \lambda x \right) \left( x \ x \ x \right)
\]
**QuickSort** is a popular and fast sorting comparative sorting algorithm with $O(n\log(n))$ complexity

- To sort list $l$, first choose a **pivot** element (arbitrary), $p$, from $l$
- Next, construct $l'$ of the elements in $l$ that are $< p$
- Also, construct $l''$ of the elements in $l$ that are $> p$
- Now, return…
  - $\text{QuickSort}(l') + [p] + \text{QuickSort}(l'')$
QuickSort is a popular and fast sorting comparative sorting algorithm with $O(n \log(n))$ complexity.

- To sort list $l$, first choose a **pivot** element (arbitrary), $p$, from $l$
- Next, construct $l'$ of the elements in $l$ that are $< p$
- Also, construct $l''$ of the elements in $l$ that are $> p$
- Now, return…
  - QuickSort($l'$) ++ [p] ++ QuickSort($l''$)

Now all sorted!

Original pivot: 65
Just returns 2
Write a function which returns the elements in a list, \( l \), which are less than some number \( n \):

\[
\text{(define (elements< } l \ n) \\
\quad \ldots)
\]

Hint: use \text{match}
Write a function which returns the elements in a list, \( l \), which are less than some number \( n \).

**Answer (one of many)**

```scheme
(define (elements< l n)
  (match l
    ['() '()]
    [`(,first ,rest ...) #:when (< first n)
      (cons first
        (elements< rest n))]
    [_ (elements< (rest l) n)])
)```
Exercise

Can also easily write `elements>`

```scheme
(define (elements< l n)
  (match l
    ['(()) '()]
    [(`(,first ,rest ...) #:when (< first n)
      (cons first (elements< rest n))]
    [_ (elements< (rest l) n)])
)
```

```scheme
(define (elements> l n)  Redundant, will fix next week
  (match l
    ['(()) '()]
    [(`(,first ,rest ...) #:when (> first n)
      (cons first (elements> rest n))]
    [_ (elements> (rest l) n)])
)
```
Exercise

Complete the definition

- To sort list l, first choose a **pivot** element (arbitrary), p, from l
- Next, construct l’ of the elements in l that are < p
- Also, construct l’’ of the elements in l that are > p
- Now, return…
  - QuickSort(l’) ++ [p] ++ QuickSort(l’’)

```
(define (quicksort l)
  (if (empty? l)
      '()
      (let* ([pivot (first l)]
             [restl (rest l)]
             [elements-lt (elements< restl pivot)]
             [elements-gt (elements> restl pivot)])
      ...))
```

69
(define (quicksort l)
  (if (empty? l)
      '()
      (let* ([pivot (first l)]
             [restl (rest l)]
             [elements-lt (elements< restl pivot)]
             [elements-gt (elements> restl pivot)])
        (append
         (quicksort elements-lt)
         (list pivot)
         (quicksort elements-gt)))))

Unfortunately, our implementation still has a bug!
Exercise: find a list l such that

\[
\text{(not (equal? (sort l <) (quicksort l)))}
\]

(define (quicksort l)
  (if (empty? l)
      '()
      (let* ([pivot (first l)]
              [restl (rest l)]
              [elements-lt (elements< restl pivot)]
              [elements-gt (elements> restl pivot)]
      (append
       (quicksort elements-lt)
       (list pivot)
       (quicksort elements-gt))))))
Our QuickSort "drops" numbers

(not (equal? (sort '(1 1) <) 
  (quicksort '(1 1))))

(define (quicksort l)
  (if (empty? l)
      '()
      (let* ([pivot (first l)]
              [restl (rest l)]
              [elements-lt (elements< restl pivot)]
              [elements-gt (elements> restl pivot)])
      (append
       (quicksort elements-lt)
       (list pivot)
       (quicksort elements-gt))))
Exercise

Solution is to make pivot a list!

(define (quicksort l)
  (if (empty? l)
    '()
    (let* ([pivot (first l)]
           [pivot-list (elements= l pivot)]
           [restl (remove pivot l)]
           [elements-lt (elements< restl pivot)]
           [elements-gt (elements> restl pivot)])
      (append
       (quicksort elements-lt)
       pivot-list
       (quicksort elements-gt))))
Observe: QuickSort recursive on data **built from** input
Thus, QuickSort uses **generative recursion**

```
(define (quicksort l)
  (if (empty? l)
      '()
      (let* ([pivot (first l)]
              [pivot-list (elements= l pivot)]
              [restl (remove pivot l)]
              [elements-lt (elements< restl pivot)]
              [elements-gt (elements> restl pivot)])
        (append
         (quicksort elements-lt)
         pivot-list
         (quicksort elements-gt))))
```
Differential / Random Testing

• Want to be very sure our code is right

• One strategy: fuzzing (“fuzz testing”)
  • Generate huge amounts of input, throw it at our code

• One issue: need to check answer is correct
  • Idea one: compare against known good version
    • This is “differential” testing
    • Sometimes want a “slow” and “fast” version
      • Slow is obviously-correct but slow
  • Idea two: just check some properties of output
    • Property-based testing
Let’s write a differential fuzzer for our QuickSort algorithm

(define (random-list i n)
  (if (= i 0)
      '()
    (cons (random 0 n)
          (random-list (- i 1) n))))

Generate random list of length i, whose elements are all in [0, n−1]
(define (counterexamples num-tries list-size max-n)
  (define (loop i l)
    (if (= i 0)
      l
      (let* ([lst (random-list list-size max-n)]
             [sorted-via-sort (sort lst <)]
             [sorted-via-qsort (quicksort lst)])
      (if (equal? sorted-via-sort sorted-via-qsort)
        (loop (- i 1) l)
        (loop (- i 1) (cons lst l))))))
  (loop num-tries '())))
Example

Compare our quicksort against Racket’s sort

(define (counterexamples num-tries list-size max-n)
  (define (loop i l)
    (if (= i 0)
        l
        (let* ([lst (random-list list-size max-n)]
               [sorted-via-sort (sort lst <)]
               [sorted-via-qsort (quicksort lst)])
          (if (equal? sorted-via-sort sorted-via-qsort)
              (loop (- i 1) l)
              (loop (- i 1) (cons lst l))))))

(loop num-tries '()))
(define (counterexamples num-tries list-size max-n)
  (define (loop i l)
    (if (= i 0)
        l
        (let* ([lst (random-list list-size max-n)]
               [sorted-via-sort (sort lst <)]
               [sorted-via-qsort (quicksort lst)])
          (if (equal? sorted-via-sort sorted-via-qsort)
              (loop (- i 1) l)
              (loop (- i 1) (cons lst l))))))
  (loop num-tries '()))

(counterexamples 300 300 1000)