How Scheme (Racket) Thinks: Lists and Recursion

September 1, 2011
CS 42: Principles and Practice of Computer Science

Assignment #1 due Monday 11:59pm
http://www.cs.hmc.edu/cs42
cs42help@cs.hmc.edu
Logging In: 1 name, 3 passwords

CIS Labs

CS Labs
(BK B102, BK B105)

http://www.cs.hmc.edu/submit

Password?
Finding Me
Recall: Picobot

Picobot

Rules

- These lines are comments.
- Remember that rules are formatted as
  # State Surroundings -> NewState
  # Picobot starts in state 0.
  # Here, state 0 goes N as far as possible
  0 xxx -> N 0  # if there's nothing to the N, go N
  0 N*** -> X 1  # if N is blocked, switch to state 1
  # and state 1 goes S as far as possible
  1 ***X -> S 1  # if there's nothing to the S, go S
  1 ***S -> X 0  # otherwise, switch to state 0

Enter rules for Picobot

Be sure to hit "Enter rules" after making changes.

Messages

OK
Alter these “up and down” rules to make the robot traverse multiple columns, moving left

0 \text{x***} \rightarrow N 0
0 \text{N***} \rightarrow X 1
1 \text{***x} \rightarrow S 1
1 \text{***S} \rightarrow X 0
Alter these “up and down” rules to make the robot traverse multiple columns, moving left.

0 x*** -> N 0
0 N*x* -> W 1
0 N*W* -> X 1

1 ***x -> S 1
1 **xS -> W 0
1 **WS -> X 0
Strategies for Problem #2?
Recall: Different interpretations of “Computing”

- Declarative Programming
- Imperative Programming
  - Functional Programming
  - Object-Oriented Programming

Prolog  **Racket**  Java  **Hmmm**

Reducing expressions to their values

more abstraction away from the machine  closer to the machine
Racket: A Functional Language

Functional Languages In General

✓ Computation model: evaluating expressions to get their result
  ▶ Minimize “side effects.”
  ▶ Generally, \( f(7) + f(7) = 2 \times f(7) \)

✓ High-Level Language: hides implementation details
  ▶ Don’t get direct access to memory
  ▶ Don’t get to decide how your data appears as bits and bytes
  ▶ Don’t need to worry about these!
  ▶ Trades programmer time for run time. (But which is more expensive?)

✓ Functions as the primary building blocks
  ▶ Toolkit approach: combine smaller functions into big ones
  ▶ Functions are ordinary (“first class”) values
Racket: A Functional Language

Racket in Particular

✓ Racket is a variant of Scheme
✓ Scheme is a “cleaned up” version of LISP, one of the very first programming languages!
✓ LISP was AI-inspired: lists are the only data structure, and almost everything is a list, including your program!
✓ Closely related to $\lambda$-Calculus, which predates computers!
Questions to Ask When Learning a New Language

✓ What kinds of data are supported? (booleans, integers, …)
✓ How do I give names to data?
✓ How do I call functions and built-in operations?
✓ How do I define my own functions?
✓ How do I make choices?
What Kinds of Data?

#f  #t
42
42.0
"this is a string"
#
characters
'year  'x
'(a b c)  '(a "hi" (42 19.0))

booleans
integers
floating-point (also: rational, complex)
strings
characters
symbols
lists
How Do I Give Names To Data?

(define answer 42)

(define howdy "hello world!")
How do I use built-in operations?

(+ 20 22)
(+ 60 (- 18))
(zero? (- 27 19))

Other Operators

and or not boolean operators
+
-
*/
modulo quotient arithmetic
max min expt mathematical functions
sqrt sin cos
How Do I Define My Own Functions?

(define (f N)
  (* N (+ N 1)))

(define (sum-upto N)
  (/ (f N) 2))
Don’t Quote Me On This!

'(f 10)

A list of length two, containing a symbol and an integer.

(f 10)

Code that runs function f, with the single input 10.
How Do You Make Choices?

\[
\text{(if } b \text{ true-branch false-branch)}
\]

Warning: what does this expression correspond to in Java/C/C++?

\[
\text{(cond [ test}_1 \text{ result}_1 \text{ ] [ test}_2 \text{ result}_2 \text{ ] [ else result}_3 \text{ ] )}
\]
Recursion

How can we take the power $b^N$?

; Given b and N, return b**N
(define (pow b N)
  (cond
    [ ] ; base case
    [else ] ; recursive case
  ))
Recursion

How can we take the power $b^N$?

; Given b and N, return b**N
(define (pow b N)
  (cond
   [            ] ; base case
    [else      ] ; recursive case
  ))

But I think we can do better!
More Efficient Code?

; Given b and N, return b**N
(define (pow b N)
  (cond
   [ (= N 0) 1 ]
   [ (odd? N) (* b (pow b (- N 1))) ]
   [ else (* (pow b (/ N 2))
               (pow b (/ N 2))) ]
  ))
More Efficient Code?

; Given b and N, return b**N
(define (pow b N)
  (cond
   [ (= N 0) 1 ]
   [ (odd? N) (* b (pow b (- N 1))) ]
   [ else (* (pow b (/ N 2))
             (pow b (/ N 2))) ]
  ))

But I think we can do better!
let is more!

let* allows you to name values temporarily:

(let* ( (y 3)
       (z 10))
  (+ y z))

Local variables, but these variables don’t vary!
; Given b and N, return b**N
(define (pow b N)
  (cond
    [ (= N 0) 1 ]
    [ (odd? N) (* b (pow b (- N 1))) ]
    [ else (let* ( (sqroot (pow b (/ N 2))) )
               (* sqroot sqroot)) ]
  )
Comments on Recursion?

“Just pretend that the function you’re writing already exists.”
Geoff Romer (CS 60)

“To understand recursion, you must first understand recursion…”
Anonymous (CS 60)
Subtlety: **let* vs. let**

**let* works sequentially, let works in parallel.**

(let* ( (n 6)  
       (m (+ n 1)) )  
(* n m))

(let ( (n 6)  
       (m (+ n 1)) )  
(* n m))

(define x 6)
(define y 7)
(let* ( (x y)  
        (y x) )  
(* x y))

(define x 6)
(define y 7)
(let ( (x y)  
        (y x) )  
(* x y))
Subtlety: `let*` vs. `let`

`let*` works sequentially.
`let` computes all the values, then creates the variables.

```
(let* ( (n 6)
        (m (+ n 1))
        (* n m))  ;; 42

(define x 6)
(define y 7)
(let* ( (x y)
        (y x))
        (* x y))  ;; 49

;; now x is 6, y is 7
```

```
(let ( (n 6)
        (m (+ n 1))
        (* n m))

(define x 6)
(define y 7)
(let ( (x y)
        (y x))
        (* x y))  ;; 42

;; now x is 6, y is 7
```
Lists

(define M '(1 (2 3) 4))

Sequential View  Recursive View

(length M)        (null? M) ; (empty? M)

(first M)         (first M) ; (car M)

(second M)        (rest M) ; (cdr M)

(third M)         (first (rest M))

(rest (rest M))   (rest (rest M)) ; '(4)

(rest (rest (rest M))) ; '()

The Fundamental List Dichotomy:
every list is empty or non-empty.
Building Lists

(define L '(h a r))
(define M '(e v))

(list M 'u 'd 'd)

(cons 'k M)

(append L L)

(reverse M)

; How to get '(h a r v e y) ?
Building Lists

(define L '(h a r))
(define M '(e v))

(list M 'u 'd 'd) ; ==> '((e v) u d d)
(cons 'k M) ; ==> '(k e v)
(append L L) ; ==> '(h a r h a r)
(reverse M) ; ==> '(v e)

; How to get '(h a r v e y) ?
Warning

(cons 5 '(2 1))

1. '(5 (2 1))
2. '(((5) 2 1))
3. '(5 2 1)
4. '(((5) (2 1)))
5. error
(cons 5 '(2 1))

1. '((5) (2 1))
2. '((5) 2 1)
3. '(5 2 1) ←
4. '(((5) (2 1))
5. error
Warning

(cons '(5) '(2 1))

1. '(5 (2 1))
2. '((5) 2 1)
3. '(5 2 1)
4. '(((5) (2 1))
5. error
Warning

(cons '(5) '(2 1))

1. '(5 (2 1))
2. '(((5) 2 1)) ←
3. '(5 2 1)
4. '(((5) (2 1)))
5. error
Warning

(list '5 '(2 1))

1. '5 (2 1))
2. '((5) 2 1)
3. '5 2 1)
4. '((5) (2 1))
5. error
Warning

(list '5 '(2 1))

1. '((5 (2 1))
2. '((5) 2 1)
3. '(5 2 1)
4. '((5) (2 1))
5. error
Warning

(list '(5) '(2 1))

1. '(5 (2 1))
2. '(((5) 2 1)
3. '(5 2 1)
4. '(((5) (2 1))
5. error
Warning

(list '(5) '(2 1))

1. '(5 (2 1))
2. '(((5) 2 1)
3. '(5 2 1)
4. '(((5) (2 1))
5. error
Warning

(append 5 '(2 1))

1. '((5) (2 1))
2. '((5) 2 1)
3. '(5 2 1)
4. '(((5) (2 1)))
5. error
(append 5 '(2 1))

1. '((5) 2 1)
2. '((5) 2 1)
3. '((5) 2 1)
4. '((5) (2 1))
5. error

Warning
Warning

(append '(5) '(2 1))

1. '(5 (2 1))
2. '((5) 2 1)
3. '(5 2 1)
4. '((5) (2 1))
5. error

(reverse '(((1 2) 3 4)))
Warning

(append '(5) '(2 1))

1. '(5 (2 1))
2. '((5) 2 1)
3. '(5 2 1)
4. '(((5) (2 1))
5. error

(reverse '(((1 2) 3 4))


Recursion: Trials (and Tribulations)

; Given a list L, return its length.
; Already built-in, but let's define it anyway.
(define (length L)
  (cond
    [ ; This is a very common pattern!
Recursion: Trials (and Tribulations)

; Given a list L, return its length.
; Already built-in, but let's define it anyway.
(define (length L)
  (cond
    [(null? L) 0]
    [else (+ 1 (length (rest L)))]))

; This is a very common pattern!
Recursion: Trials (and Tribulations)

; Given a value e and a list L, check if e is in L
; Already built-in, but let's define it anyway.
(define (member e L)
  (cond
   [ ; the built-in member returns a list, rather than #t
   [ ; Note: for "if" and "cond", anything that is not #f
     [ ; is considered true, including lists (and the empty list!)
     ]
   ]
  )
)
Recursion: Trials (and Tribulations)

; Given a value e and a list L, check if e is in L
; Already built-in, but let's define it anyway.
(define (member e L)
  (cond
   [ (null? L) #f ]
   [ (equal? e (first L)) #t ]
   [ else (member e (rest L)) ]))

;; the built-in member returns a list, rather than #t

;; Note: for "if" and "cond", anything that is not #f
;; is considered true, including lists (and the empty list!)
“Quiz” 2

; Given L, return same list but with
; top-level elements in opposite order
(define (reverse L)
  (cond
   ...
  ))

; What do I compute?
(define (mystery N)
  (cond
   ...
  ))

; Given L, flatten out all nesting
; e.g., (flatten '(1 (2 3 (4)) (5 6)))
; ==> '(1 2 3 4 5 6)
(define (flatten L)
  (cond
   ...
  ))

; given N>1, compute the smallest prime
; factor of N, e.g., (spf 45) ==> 3
(define (spf N)
  (cond
   ...
  ))
“Quiz” 2 Solutions

; Given L, return same list but with
; top-level elements in opposite order
(define (reverse L)
  (cond
   [ (null? L) L ] ;; or return '()
   [ else (append (reverse (rest L))
                (list (first L))) ]
  ))

Notes:
✓ We can’t use cons to put a single item at the end of a list.
✓ Append expects two lists, not a list and a single element. (That’s why we have to build a list of length one, containing only first L):

(reverse '(1 2 3)) ===> append (reverse '(2 3)) '(1)
====> append '(3 2) '(1)
====> '(3 2 1)
“Quiz” 2 Solutions

; Given L, flatten out all nesting
; e.g., (flatten '(1 (2 3 (4)) (5 6)))
; ==> '(1 2 3 4 5 6)
(define (flatten L)
  (cond
    [ (null? L) L ]
    [ (not (list? L)) (list L) ]
    [ else (append (flatten (first L)) (flatten (rest L))) ]
  )
)
"Quiz" 2 Solutions

; What do I compute?
(define (mystery N)
  (cond
    [ (< N 2) 0 ]
    [ else (+ 1
              (mystery (quotient N 2)))]
  ))

The number of times \(N\) can be divided by 2, throwing away remainders, until we get below 2. That is, (an integer approximation to) the logarithm base 2 of \(N\).
Putting `assoc` in it

An association list (a-list) is a list of pairs.
It’s a simple way to associate keys with values
(like dictionaries, lookup tables, …).

```racket
(define rome '( (#\I 1) (#\V 5) (#\X 10) (#\L 50)
                  (#\C 100) (#\D 500) (#\M 1000) ))
```

```racket
(assoc #\D rome)  ; ==> '(#\D 500)
(assoc #\Z rome)  ; ==> #f
```

;; Note: for "if" and "cond", anything that is not #f
;; is considered true, including lists (and the empty list!)