Login

Bonus Assignment: Store-Passing Style

> Trolls' purses are the mischief, and this was no exception. " 'Ere, 'oo are you?" it squeaked, as it left the pocket.

Introduction

Place all of your code in a file named `bonus.rkt`, and submit it via Canvas – if there's no page for submission, post on piazza.

Make use of the macros and SPS material from class this week, and you might consult the macro tutorials below.

For SPS, you could also consult _Essentials of Programming Languages_. For macros, you should probably look through Syntax rules for the merely eccentric [http://www.phyast.pitt.edu/~micheles/syntax-rules.pdf]. If you're still hungry for more after that, try Syntax rules for the mildly insane [http://petrofsky.org/src/primer.txt].

You probably (maybe?) used `match-let` in class to implement SPS. On your homework, we'll be using `values` and `let-values`. These operate similarly.

Assignment

Part 1 SPS

Your task is to transform the functions below into store-passing style such that they have the correct semantics described. **To implement SPS, you will use values [http://docs.racket-lang.org/reference/values.html?q=values#%28def._%28%28quote._~23~25kernel%29._values%29%29] and let-values [http://docs.racket-lang.org/reference/let.html?q=let-values#%28form._%28%28quote._~23~25kernel%29._let-values%29%29]. Consult lecture notes or office hours if you forget how these work.**

1 Recall `filter` from assignment 1 [https://www.cs.indiana.edu/cgi-pub/c311/doku.php?id=assignment-1]. It would be useful if all of the results removed from the list were also handed back to us. That way, we could `partition` the input based on a predicate. Implement `filter-sps` that does exactly that.

```
> (filter-sps even? '(1 2 3 4 5 6 7 8 9 10) '())
(2 4 6 8 10)
(1 3 5 7 9)

> (filter-sps odd? '(1 2 3 4 5 6 7) '())
(1 3 5 7)
(2 4 6)

> (filter-sps (lambda (x) (or (> x 6) (< x 2))) '(1 2 3 4 5 6 7) '())
(1 7)
(2 3 4 5 6)
```

1 Consider a function `filter*` which performs filter on (potentially) deep lists as follows:

```
(define filter*
  (lambda (f ls)
    (cond
      [(null? ls) '()]
      [(list? (car ls))
       (cons (filter* f (car ls)) (filter* f (cdr ls)))]
      [(f (car ls)) (cons (car ls) (filter* f (cdr ls)))]
      [else (filter* f (cdr ls))])))

> (filter* even? '(1 2 3 4 5 6))
(2 4 6)

> (filter* odd? '(1 (2 3 (4 5)) 6 7))
(1 (3 (5)) 7)

> (filter* (lambda (x) (or (even? x) (< 7 x))) '(1 (2 3 (4 5)) 6 7 ((8 9) 10)))
((2 (4)) 6 ((8 9) 10))
```

Produce a function `filter*-sps` that performs the same function as `filter-sps`, but works on deep lists and preserves the structure of each.

```
> (filter*-sps even? '(1 2 3 4 5 6) '())
(2 4 6)
(1 3 5)

> (filter*-sps odd? '(1 (2 3 (4 5)) 6 7) '())
(1 (3 (5)) 7)
((2 (4)) 6)

> (filter*-sps (lambda (x) (or (even? x) (< 7 x))) '(1 (2 3 (4 5)) 6 7 ((8 9) 10)) '())
((2 (4)) 6 ((8 9) 10))
(1 (3 (5)) 7 (()))
```

2 It is possible to exploit store-passing style for greater efficiency in programs. One such usage is memoization [http://en.wikipedia.org/wiki/Memoization] of results, which can greatly speed up computation. Implement `fib-sps` that uses the store for memoization as follows. (The answers below are written with the `.` explicitly, this is simply to clarify the results)

```
> (fib-sps 0 '())
0
((0 . 0))

> (fib-sps 1 '())
1
((1 . 1))

> (fib-sps 3 '())
2
((3 . 2) (2 . 1) (0 . 0) (1 . 1))

> (fib-sps 10 '())
55
((10 . 55)
 (9 . 34)
 (8 . 21)
 (7 . 13)
 (6 . 8)
```

```
 (5 . 5)
 (4 . 3)
 (3 . 2)
 (2 . 1)
 (1 . 1)
 (0 . 0))
```

Depending upon your implementation, your store may not print in precisely this order. That is acceptable. But to ensure your program is correct, ensure that you arent' duplicating any subcomputations, and that you have all the subcomputations.

Part 2 Macros

Let's implement some macros. For the following questions, make sure to use only `syntax-rules` macros, as we wrote in class. For the `and*` and `list*` problems, it is *not* acceptable to use the `and` and `list*` built into Racket in your implementations. Also, don't use `match` in your solutions.

Note: As you work on the following problems, you can use the macro stepper in DrRacket, `(syntax→datum (expand <syntax expr>))` and `expand-only` as we might have in class/lab, or some of the utilities found here [http://docs.racket-lang.org/reference/Expanding_Top-Level_Forms.html] to see what a macro expands to.

```
> (syntax->datum (expand '(cond (#t #f) (else 7))))
(if '#t (let-values () '#f) (let-values () '7))
```

3. `and*` This should work similarly to Racket's `and`.

```
> (and* 1 2 3)
3
> (and* #f)
#f
> (and*)
#t
> (and* 'a)
a
> (and* #t #t #t #t #t #t #t #t #f)
#f
```

4. `list*`

`list*` cons-es together its arguments. If the final argument is not a list, `list*` should return an improper list. If a single argument is passed, it should simply return that argument. When called with no arguments, your macro should report an error by calling: `(raise-syntax-error "Incorrect argument-count to list*")`. Your answer should operate similarly to Racket's `list*`.

```
> (list* 'a 'b 'c 'd)
(a b c . d)
> (list* 'a)
a
```

5. macro-list

The Racket function `list` can be implemented simply as a function in Racket using _variadic_ (n-ary) lambdas.

```
> (define list (lambda a a))
> (list 1 2 3 4)
```

Note the absence of parentheses around the formal parameter to the function. But in the early days of Lisp, there were no variadic functions. Instead, `list` was implemented as a recursive macro. Implement `macro-list`, which takes any number of arguments and builds a list of them.

```
> (macro-list)
()
> (macro-list 1 'b 2 'd)
(1 b 2 d)
```

6. mcond

We know that we can treat cond as a series of `if` statements. Write an `mcond` macro which acts like cond, but desugars to a series of nested `if`s. Make sure to provide the appropriate treatment for `else` clauses.

```
> (mcond
    (#f #t)
    (else 'dog))
dog
> (mcond
    (else 'cat))
cat
> (mcond
    (#t #t)
    (unbound variables))
#t
```

Standard Racket `cond` has a good deal of extra behavior, including support for one-element clauses, ⇒ notation, multiple bodies in a clause, etc. You aren't required to implement this behavior, but if the mood strikes you then go hogwild.

```
> (cond (#t 'a 'b 'c))
c
> (cond
    (#f 'a-thing)
    (#f)
    ('turtle)
    (else 'rock))
turtle
> (cond
    ((member 'a '(d a g w o o d)) => length)
    (else 'not-a-member))
6
```

7. Macro-map.

Notice that we cannot use `map` with a macro:

```
> (map (lambda (x) (list x x)) '(a b c))
((a a) (b b) (c c))
> (define-syntax copy-code
    (syntax-rules ()
      [(_ x) `(,x x)]))
> (copy-code (lambda (x) x))
(#<procedure> (lambda (x) x))
> (copy-code 'a)
(a 'a)
> (map copy-code '(a b c))
stdin::167: copy-code: bad syntax
  in: copy-code
  context...:
```

Macros cannot be passed as arguments to a function in that manner. Instead, they have to be expanded from matches of a pattern into some new template. To get around this problem, let's define a macro `macro-map`, which *will* allow us to map a macro.

```
> (macro-map quote '((trinidad and tobago) (saint vincent and the grenadines) (antigua and barbuda)))
((trinidad and tobago)
 (saint vincent and the grenadines)
 (antigua and barbuda))
> (macro-map copy-code '((lambda (x) x) (lambda (x) (+ 2 x)) (lambda (x) 7)))
((#<procedure> (lambda (x) x))
 (#<procedure> (lambda (x) (+ 2 x)))
 (#<procedure> (lambda (x) 7)))
> (define-syntax quote-quote
    (syntax-rules ()
      [(_ e) (quote (quote e))]))
> (macro-map quote-quote '((trinidad and tobago) (saint vincent and the grenadines) (antigua and barbuda)))
('(trinidad and tobago)
 '(saint vincent and the grenadines)
 '(antigua and barbuda))
```

Brainteaser

Omitted. No worries this week.

Just Dessert

8. condre

There are some unfortunate limitations to `cond`. For instance, how many times have you had to do something like the following:

```
(define proc-list
  (lambda (ls)
    (cond
      ((null? ls) '())
      (else ;; This part here.
        (let ((a (car ls)))
          (cond
            ((number? a) (cons (even? a) (proc-list (cdr ls))))
            ((boolean? a) (cons a (proc-list (cdr ls))))
            (else (cons #f (proc-list (cdr ls))))))))))
```

The part to which I want to direct your attention is that else-let-cond component. I really want it to be all part of one cond block, but I have to add an else, introduce a let-binding, then start a new cond under a new indentation. Yuk. Enter `condre` (for Andre's `cond`).

```
> (condre
    (let ((a 5))
      ((number? a) #t)
      (else #f)))
#t
> (define proc-list
    (lambda (ls)
      (condre
        ((null? ls) '())
        (let ((a (car ls)))
          ((number? a) (cons (even? a) (proc-list (cdr ls))))
          ((boolean? a) (cons a (proc-list (cdr ls))))
          (else (cons #f (proc-list (cdr ls))))))))
> (proc-list '(#t 2 3 #f 'dog))
(#t #t #f #f #f)
```

Andre, being Andre, added support for `let*`, `letrec`, `letrec*`, ⇒, 1 or more bodies, and more than one body on the right-hand side of an arrow.

```
> (condre
    (#f)
    (letrec ((ls? (lambda (ls)
                    (condre
                      [(null? ls) #t]
                      [(pair? ls) (ls? (cdr ls))]
                      [else #f]))))
      ((ls? '(1 2 3 . 4)) #f)
      ((ls? '(1 2 3 4)) => (display 'Should-be-seven->) (lambda (t) (set! t 7) t))
      (else #f)))
Should-be-seven->7
```

Obviously, this is just for fun, so you can include just as much of this as you feel compelled to do. But you don't wanna be walking around with an off-brand `condre`, do you?

Extra bonus dessert

With or without peeking implement the interpreter as a `syntax-rules` macro. That is, by the time macro expansion is finished, you'll have completed the evaluation of the program. You already have a document that shows how to do this, but it's more fun if you can do it with a minimum of peeking. You'll need continuation-passing macros, perhaps accumulator-style macros, and some of the Petrofsky/Kiselyov trickery to do some of the tests you'll need with pattern matching.