Review: “Continuation-Passing Style”

All tail positions are continuations or recursive calls

(define witness-cps (p? xs succ fail)
  (if (null? xs)
    (fail)
    (let ([x (car xs)])
      (if (p? x)
        (succ x)
        (witness-cps p? (cdr xs) succ fail))))

Compiles to tight code
Solving Boolean formulas

A formula is one of these:

- Symbol (stands for a variable)
- Record (make-not \( f \)); \( f \) is a formula
- Record (make-or \( fs \)); \( fs \) is a list of formulas
- Record (make-and \( fs \)); \( fs \) is a list of formulas

In context of:

(record not [arg])
(record or [args])
(record and [args])
Your turn: Find satisfying assignment!

(val f1 (make-and (list4 'x 'y 'z (make-not 'x)))))
; x \( y \) \( z \) \( !x \)

(val f2 (make-not (make-or (list2 'x 'y)))))
; x \( y \)

(val f3 (make-not (make-and (list3 'x 'y 'z)))))
; !(x \( y \) \( z \))
Wait for it ...
Satisfying assignments

(val f1 (make-and (list4 'x 'y 'z (make-not 'x))))
; x \(\land\) y \(\land\) z \(\land\) !x ;; NONE

(val f2 (make-not (make-or (list2 'x 'y))))
; x \(\lor\) y ;; \{ x \rightarrow \#f, y \rightarrow \#f \}

(val f3 (make-not (make-and (list3 'x 'y 'z))))
; !(x \(\land\) y \(\land\) z) ;; \{ x \rightarrow \#f, ... \}
Find assignment using continuations

```
start +-------------------+ succeed
       |                   |
-------|---------------------|
       |    | solver |        |
fail   |---------------------|<-------
       |                   | fail
       |                   |<-------
succeed +-------------------+ resume
```

**start** Gets **partial solution**, **fail**, **succeed**
(On homework, “solution” is assignment)

**fail** Partial solution won’t work (no params)

**succeed** Gets improved solution + **resume**

**resume** If improved solution won’t work, try another (no params)

A composable unit!
Continuations for the solver

A big box contains two smaller boxes A and B

There are two ways to wire them up (board)

Imagine A and B as formulas

Imagine A as a formula, B as a list of formulas!
Solving a literal

; (satisfy-literal-true x current succ fail) =
; (succ current fail), when x is bound to #t in cur
; (fail), when x is bound to #f in cur
; (succ (bind x #t current) fail), x unbound in cur

(define satisfy-literal-true (x current succ fail)
  (if (bound? x current)
      (if (find x current)
          (succ current fail)
          (fail))
      (succ (bind x #t current) fail)))
Lisp and Scheme Retrospective
μScheme and the Five Questions

Abstract syntax: imperative core, let, lambda

Values: S-expressions
   (especially cons cells, function closures)

Environments: A name stands for a mutable location holding value

Evaluation rules: lambda captures environment

Initial basis: yummy higher-order functions
Real Scheme: Macros

A Scheme program is *just another S-expression*

- Function `define-syntax` manipulates syntax at compile time
- Macros are *hygienic*—name clashes impossible
- `let`, `&&`, `record`, others implemented as macros

(See book sections 2.16, 2.17.4)
Real Scheme: Conditionals

(cond [c1 e1] ; if c1 then e1
     [c2 e2] ; else if c2 then e2
     ...        ...  
     [cn en]) ; else if cn then en

; Syntactic sugar—'if' is a macro:
(if e1 e2 e3) == (cond [e1 e2]
                      [#t e3])
Real Scheme: Mutation

Not only variables can be mutated.

Mutate heap-allocated cons cell:

\[(\text{set-car! } '(a \ b \ c) \ 'd) \Rightarrow (d \ b \ c)\]

Circular lists, sharing, avoids allocation
• still for specialists only