## **Review: Tail calls**

#### Which functions are called in tail position?

```
(define split-list (originals)
  (list2 (every-other-element originals)
         (if (null? originals)
             '()
             (every-other-element (cdr originals)))))
(define use-macro (lhs abbrevs expand error)
  (if (null? abbrevs)
      (error lhs)
      (if (= lhs alist-first-key (abbrevs))
          (expand (alist-first-attribute abbrevs))
          (use-macro lhs abbrevs expand error))))
```

## **Review: Tail calls**

#### Which functions are called in tail position?

```
(define split-list (originals)
  (list2 (every-other-element originals)
         (if (null? originals)
             '()
             (every-other-element (cdr originals)))))
(define use-macro (lhs abbrevs expand error)
  (if (null? abbrevs)
      (error lhs)
      (if (= lhs alist-first-key (abbrevs))
          (expand (alist-first-attribute abbrevs))
          (use-macro lhs abbrevs expand error))))
```

### **Expressions to avoid**

(if #t #f) ; worst

(append (list1 <x>) <ys>) ; most common

(cons <x> (cons <y> '())) ; prefer `list2`

# **Homework alert**

### **Functions list-of, formula?**

- Can be passed any value
- Must handle all cases (Figure 2.1, page 95)

## **Review: "Continuation-Passing Style"**

#### All tail positions are continuations or recursive calls

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

**Compiles to tight code** 

# Homework: 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

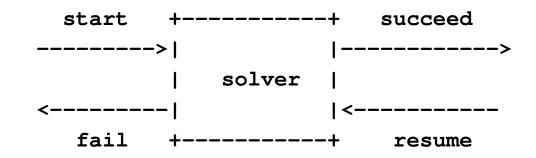
- (val f1 (make-and (list4 'x 'y 'z (make-not 'x))))
  ; x /\ y /\ z /\ !x ;; NONE
- (val f2 (make-not (make-or (list2 'x 'y))))
  ; !(x \/ y) ;; { x |-> #f, y |-> #f }
- (val f3 (make-not (make-and (list3 'x 'y 'z)))
  ; !(x /\ y /\ z) ;; { x |-> #f, ... }

# Finding a satisfying assignment

**Example formula:** 

 $(x \setminus / y) / (!x / z)$ 

# Find assignment using continuations



- 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**

# **Five powerful questions**

- 1. What is the abstract syntax? Syntactic categories? Terms in each category?
- 2. What are the values? What do expressions/terms evaluate to?
- 3. What environments are there? What can names stand for?
- 4. How are terms evaluated? Judgments? Evaluation rules?
- 5. What's in the initial basis? Primitives and predefined, what is built in?

## $\mu$ Scheme and the Five Questions

Abstract syntax: expressions and definitions imperative core, let, lambda

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

Environments: A name stands for a mutable location holding a value

**Evaluation rules:** lambda captures environment

**Initial basis: yummy higher-order functions** 

# **Full 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)

### **Full Scheme: Conditionals**

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

## **Full Scheme: Mutation**

Not only variables can be mutated.

Mutate heap-allocated cons cell:

(set-car! '(a b c) 'd) => (d b c)

**Circular lists, sharing, avoids allocation** 

still for specialists only