Expressions to avoid

```
(if <p> #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 2.2, page 94)**

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

Solving Boolean formulas with continuations

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

given the following `record` definitions (in book):

```
(record not [arg])
(record or  [args])
(record and [args])
```

Solving Boolean Formulas

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

Answer: NONE

Solving Boolean Formulas

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

Answer:

```
{ x |-> #f, y |-> #f }
```

Solving Boolean Formulas

```
(val f3 (make-or (list2 'x 'y)))
   ; x \/ y
```

Answer: Three solutions!

```
{ x |-> #t, ... }
{ y |-> #t, ... }
```

Solving Boolean Formulas

```
(val f4 (make-not (make-and (list3 'x 'y 'z))))
   ; !(x /\ y /\ z)
```

Answer: Many solutions

(all 7 ways the variables can not all be `#t`):

```
{ x |-> #f, ... }, ...
```

Finding a satisfying assignment

Example formula:

```
(x \/ y) /\ (!x /\ !z)
```

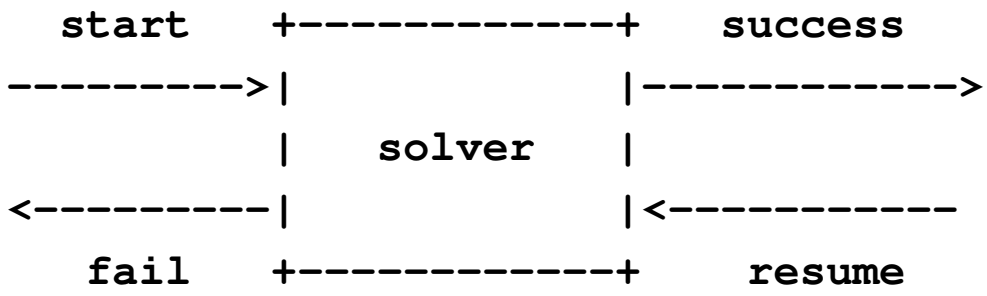
Problem Decomposition

	Goal: True	Goal: False
x	Check `current`, maybe extend	
$\neg$ A	Make A False	Make A True
A $\wedge$ B	Both	Either
A $\vee$ B	Either	Both

`current` is partial assignment of truth values to variables.

Compose a network of solvers using continuations

Formula types: `x`, `!x`, `A ∧ B`, `A ∨ B`



start	**Passed `current` (partial solution), `fail`, `success`**
`success`	**Call with extended solution + `resume`**
`resume`	**Called by downstream solver with no params if extended solution won't work**
`fail`	**Call with no params if `current` won't work**

A composable unit!

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)))
```

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!

Lisp and Scheme Retrospective

Five powerful questions

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

μScheme and the Five Questions

Values:

S-expressions (esp. `cons` cells and closures)

Environments:

A name stands for a mutable location holding a value

Abstract syntax:

Expressions and definitions, imperative core, `let`, `lambda`

Evaluation rules: `lambda` captures environment

Initial basis: powerful higher-order functions

Next steps

Before tomorrow's lecture,

Consider good and bad points of Scheme

(Informed by your experience)

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

; Syntactic sugar---'if' is a macro:
(if e1 e2 e3) == (cond [e1 e2]
                       [#t e3])
```

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