Predicates – procedures that answer #t or #f – allow us to test whether some condition holds.

Conditional expressions enable us to make choices based on the result of our test.

E.g., “If it’s raining, I’ll take an umbrella; otherwise, I won’t.”

Finding the larger of two numbers

(define maximum
  (lambda (x y)
    (if (>= x y) x y)))

> (maximum 17 23)
23
> (maximum (- 36 3) (- 10 13))
33
Finding the smaller of two numbers

```scheme
(define minimum
 (lambda (x y)
   (if (<= x y) x y)))
```

> (minimum 17 23)
17

> (maximum (- 36 3) (- 10 13))
-3

The if special form

```scheme
(if <test condition>
    <then expression>
    <else expression>)
```

Evaluating an if special form:
First, evaluate <test condition>.
If it evaluates to anything other than #f, evaluate the <then expression> and return the result.
Otherwise, evaluate the <else expression> and return the result.

Why is (if ...) considered a special form?
Scheme always evaluates the condition.
Depending on the value of the condition, Scheme evaluates either the consequent or the alternative, but not both.

Suppose “if” were the name of a procedure?
We would expect Scheme to evaluate all the arguments, i.e., Scheme would evaluate both the consequent and the alternative.

Why does this matter?

> (define foo
  (lambda (x)
    (if (= x 0) x (/ 1 x))))

> (foo 0)
0

Suppose “if” were the name of a procedure.
An attempt to evaluate (foo 0) would lead to a division by zero error.
**Type predicates**

Used to test the type of a data object.

For each type of data, we have a corresponding type predicate.

### number?

```lisp
> (number? 7)
#t
> (number? (- 10 5))
#t
> (number? 'zaphod)
#f
> (number? '(marvin eddie))
#f
```

### symbol?

```lisp
> (symbol? 'zaphod)
#t
> (symbol? (car '(marvin eddie)))
#t
> (symbol? 7)
#f
> (symbol? '(arthur trillian))
#f
```

### null?

null? returns #t if its argument is the empty list.

null? returns #f otherwise.

```lisp
> (define empty-list '())
> (null? empty-list)
#t
> (null? '())
#t
> (null? 0)
#f
```
The `procedure?` predicate

```scheme
> (procedure? first)
#t
> (procedure? (lambda (x) (* x x)))
#t
> (procedure? 'first)
#f
> (procedure? '(lambda (x) (* x x)))
#f
```

```scheme
> (if (number? 3) 'yes 'no)
yes
> (if (number? 'x) 'yes 'no)
no
```

```
(define careful-mult-by-10
  (lambda (x)
    (if (number? x)
      (* x 10)
      'not-a-number)))
```

```scheme
> (careful-mult-by-10 35)
350
> (careful-mult-by-10 #t)
not-a-number
> (careful-mult-by-10 ())
not-a-number
```

Defining the `type-of` function

`type-of` will take any data as its argument. `type-of` will return a symbol, indicating the data type of the argument.

E.g.,

```scheme
> (type-of 9)
'number
> (type-of '())
'empty-list
```
if special forms can be nested, but doing so can make your code hard to read:

```scheme
(define type-of
  (lambda (item)
    (if (null? item)
        'empty-list
      (if (list? item)
          'list
        (if (number? item)
            'number
          (if (symbol? item)
              'symbol
            (if (procedure? item)
                'procedure
              'other)))))))
```

A better way...

```scheme
(define type-of
  (lambda (item)
    (cond ((null? item) 'empty-list)
          ((list? item) 'list)
          ((number? item) 'number)
          ((symbol? item) 'symbol)
          ((procedure? item) 'procedure)
          (else 'other)))))
```

The **cond** special form

The cond special form is very convenient when you have several conditions you want to test. If you find yourself writing nested if expressions, consider switching to a cond.

Evaluating the cond special form:

Each test expression is evaluated until one, say, `<test expression m>` evaluates to something other than `#f`.

No further test expressions are evaluated.

The expressions `<expression m.1>`, `<expression m.2>`, … are then evaluated in turn.

The value of the last expression in that sequence is returned as the value of the cond expression.
(cond (<test expression 1>
  <expression 1.1> <expression 1.2> ...)
  (<test expression 2>
  <expression 2.1> <expression 2.2> ...)
  ...
  (<test expression n>
  <expression n.1> <expression n.2> ...))

The last expression, i.e., <test expression n>, should be one of the following: #t or the else keyword.

This ensures that the last test expression is always true, letting you give a default case.

(define classify
  (lambda (x)
    (cond ((< x 0) 'neg)
          ((= x 0) 'zero)
          ((< x 10) 'small)
          (else 'big)))))

> (classify 3)
small
> (classify -1)
neg
> (classify 0)
zero
> (classify 34)
big

> (cond (#f 1)
    (#f 2)
    (#t 3)
    (#f 4)
    (else 5))
3
> (cond (#f (printf "1..."))
    (#f (printf "2..."))
    (printf "3..."))
    (#f (printf "4..."))
    (#t (printf "TRUE!!!")
      (newline))
    (#f (printf "5..."))
    (#t (printf "Second true!")))
TRUE!!!

Another useful predicate is integer?

(define divides-evenly?
  (lambda (x y)
    (integer? (/ x y)))))

> (divides-evenly? 3 4)
#f
> (divides-evenly? 12 3)
#t
The not predicate
(not <expression>)

not takes one argument, which is normally a Boolean expression.

not returns #t if its argument evaluates to #f.
not returns #f otherwise.

> (not #t)
#f
> (not #f)
#t
> (not 3)
#f
> (not (not #f))
#f
> (equal? '(superman)  
     (cons 'superman '()))
#t
> (not (equal? '(superman)  
     (cons 'superman '())))
#f

The and special form
(and <expression 1>  
 <expression 2>  
 ...  
 <expression n>)

Evaluating the and special form:

Scheme evaluates the conditions in order.
If any condition evaluates to #f, then Scheme immediately returns #f as the value of the entire expression.
Otherwise, then Scheme returns the result of <expression n> as the value of the entire and expression.
Remember that non-\texttt{#f} values count as true!

\[
> \text{(and} \; 1 \; 2 \; \text{#f} \; 3 \; 4) \\
\text{#f}
\]

It doesn't print the error messages because it stops as soon as \texttt{#f} is encountered.

This means we don't need to worry about generating a divide-by-zero error in the following:

\[
(\text{and} \; \text{#f} \; (/ \; 1 \; 0))
\]

\section*{The or special form}

\begin{verbatim}
(\text{or} \; <expression_1> \\
 <expression_2> \\
 ... \\
 <expression_n>)
\end{verbatim}

Similar to the \texttt{and} special form, but the idea is to return true (or something that counts as true) if at least one of the expressions counts a true.

Scheme evaluates the conditions in order:

- If any condition evaluates to \texttt{#t} or something that counts as true, then Scheme immediately returns it as the value of the entire or expression.
- If all conditions evaluate to \texttt{#f}, then Scheme returns \texttt{#f} as the value of the entire or expression.

\section*{Acknowledgments}

This lecture incorporates material from:

- Tom Ellman
- Luke Hunsberger