Functions that take functions as arguments

✔ `funcall` is a function that applies its first argument to the rest of its arguments

```
(funcall fn arg1 arg2 ... )
```

✗ the function *fn* is applied to the arguments *arg1, arg2, ...*

✗ `fn` can evaluate to a function object, or to a symbol which has a function definition:

```
USER: '+
+

USER: #'+
#<Function + @ #x504e36>

USER: (funcall '+ 1 2 3)
6

USER: (funcall #'+ 1 2 3)
6
```

✔ think of `(funcall fn arg1 arg2 ... )` as equivalent to

```
(eval (list fn arg1 arg2 ... ))
```

✔ this is different from

```
(fn arg1 arg2 ...)
```

because here `fn` is not *evaluated*

Functions that take functions as arguments

- `apply` is a function that applies its first argument to the elements of its second argument

  (apply *fn arglist*)

  - the function *fn* is applied to the elements of the argument list *arglist*
  - `fn` can evaluate to a function object, or to a symbol which has a function definition:

```
USER: '+
+

USER: #'+
#<Function + @ #x504e36>

USER: (apply '+ '(1 2 3))
6

USER: (apply #'+ '(1 2 3))
6
```

- think of (apply *fn arglist*) as equivalent to

  (eval (cons *fn arglist*))

- this is different from

  `(fn arg1 arg2 ...)`
  because here `fn` is not *evaluated*

Mapping functions

- ✔ `mapcar` can apply a function of one argument to each element of a list in succession, and returns a list of the results

- ✔ `mapcar` can (almost...) be defined as

```
;;; map a function over successive car's of a list
;;;
(defun mapcar (fn list)
    (if (endp list) '()
        (cons
            (funcall fn (car list))
            (mapcar fn (cdr list)))))
```

```
USER: (mapcar #'evenp '(1 2 3 4 5 6 7))

(NIL T NIL T NIL T NIL)
```

- ✔ in fact, `mapcar` can apply a function of several arguments to the elements of several lists in succession, returning a list of the results

```
USER: (mapcar #'list '(1 2 3 4) '(5 6 7 8) '(9 10 11 12))

((1 5 9) (2 6 10) (3 7 11) (4 8 12))


USER: (mapcar #'expt '(1 2 3 4) '(4 3 2 1))

(1 8 9 4)
```

Mapping functions

- mapping a function over a sequence of arguments is a central idea in functional programming

- mapping can be used in place of iteration

```
(defun first-n-integers (n)
    (if (zerop n) NIL
        (append (first-n-integers (- n 1)) (list n))))
```

```
USER: (first-n-integers 5)
(1 2 3 4 5)

USER: (mapcar #'factorial (first-n-integers 5))
(1 2 6 24 120)
```

- suppose we want a list of the first 5 powers of 2. We can define

```
(defun power-of-2 (n)
    (expt 2 n))
```

```
USER: (mapcar #'power-of-2 (first-n-integers 5))
(2 4 8 16 32)
```

- but we can also do this without defining the new function power-of-2...

Creating anonymous functions with lambda

- LISP is based on the mathematical formalism of the “lambda calculus” (A. Church, 1936) in the theory of computation
- In the lambda calculus, a formula (for example)
  `λxy. x+y+1`
  denotes “a function of two arguments which gives their sum plus one”
- In LISP, this example corresponds to a lambda expression
  `(lambda (x y) (+ x y 1))`
- The lambda expression can be used anywhere a function can be used

```
USER: ((lambda (x y) (+ x y 1)) 2 3)
6

USER: (funcall #'(lambda (x y) (+ x y 1)) 2 3)
6

USER: (funcall '(lambda (x y) (+ x y 1)) 2 3)
6

USER: (setq f #'(lambda (x y) (+ x y 1)))
#<Interpreted Function (unnamed) @ #xa682ee>

USER: (funcall f 2 3)
6

USER: (f 2 3)
Error: attempt to call `F' which is an undefined function
```

Creating anonymous functions with lambda

- ✔ *lambda expressions* are what LISP uses to represent interpreted function definitions

```
USER: (defun f (x y) (+ x y 1))
F

USER: (f 2 3)
6

USER: (function-lambda-expression #'f)
(LAMBDA (X Y) (BLOCK F (+ X Y 1)))
```

- ✔ lambda expressions can be convenient as arguments to functions that require functions as arguments ... (whew!)

```
USER: (mapcar #'(lambda (n) (expt 2 n)) '(1 2 3 4 5))
(2 4 8 16 32)
```

- ✔ lambda expressions can be convenient as values to return from functions that return functions...

lambda expressions

- ✔ lambda expressions can be convenient as values to return from functions that return functions
- ✔ Example: define a function `make-power-function`
  - ✗ `make-power-function` is a function of one argument, n
  - ✗ it returns a function of one argument, x
  - ✗ the returned function computes x^n

```
(defun make-power-function (n)
   #'(lambda (x) (expt x n)))

USER: (setq square (make-power-function 2))
#<Interpreted Closure (:INTERNAL MAKE-POWER-FUNCTION)..>

USER: (funcall square 5)
25

USER: (setq cube (make-power-function 3))
#<Interpreted Closure (:INTERNAL MAKE-POWER-FUNCTION)..>

USER: (funcall cube 5)
125
```

- ✔ `#'` creates a *lexical closure*: lexical bindings keep their values

Note that

```
(defun make-power-function (n)
   '(lambda (x) (expt x n)))
```

would not work as desired

More about parameter lists for functions

- Parameter lists are also called “lambda lists”
- Paramter lists can get very fancy...

lambda syntax

p. 76

CLtL II

Optional parameters

- Optional parameters can be specified in a function's parameter list
- Optional parameters follow required parameters (if any)
- Optional parameters are bound to NIL if actual arguments are not supplied in the function call, unless defaults are given

```
(defun foo (a x q &optional b z)
    (list a x q b z))

USER: (foo 1 2 3)
(1 2 3 NIL NIL)

USER: (foo 1 2 3 4)
(1 2 3 4 NIL)

USER: (foo 1 2 3 4 5)
(1 2 3 4 5)
```

- Optional parameters can have defaults specified
- Optional parameters are bound to the defaults if actual arguments are not supplied in the function call
- Defaults may refer to required or optional parameters earlier in the parameter list

```
(defun baz (a x q &optional (b 77) (z x))
    (list a x q b z))

USER: (baz 1 2 3)
(1 2 3 77 2)
```

Keyword parameters

- Keyword parameters can be specified in a function's parameter list
- Keyword parameters follow required parameters (if any)
- Keyword parameters without defaults are bound to NIL if actual arguments are not supplied in the function call
- Keyword parameters can have defaults supplied as for optional parameters
- In the function call, arguments corresponding to keyword parameters can appear in any order after the required parameters, but...
- In the function call, arguments corresponding to keyword parameters must appear paired with their keyword, which is the parameter name with a : prefixed

```
(defun foo (a x q &key b z)
    (list a x q b z))

USER: (foo 1 2 3)
(1 2 3 NIL NIL)

USER: (foo 1 2 3 :z 4 :b 5)
(1 2 3 5 4)

USER: (foo :b 4 1 :z 5)
(:b 4 1 NIL 5)
```

- Keyword parameters can save having to remember parameter ordering
- Many built-in LISP functions have keyword parameters:

```
(open "myfile" :direction :output :element-type 'character
    :if-exists :overwrite)
```

Rest parameter

- A rest parameter can be specified in a function's parameter list
- The rest parameter is bound to a list containing arguments not already assigned to required parameters
- The rest parameter provides a way to define functions which can take any number of arguments

```
(defun foo (a b &rest r)
    (list a b r))

USER: (foo 1 2 3 4 5 6)
(1 2 (3 4 5 6))

USER: (foo 1 2)
(1 2 NIL)

(defun my-list-function (&rest result)
    result)

USER: (my-list 1 2 3 4 5 6)
(1 2 3 4 5 6)


(defun applicable-and (&rest args)
    (cond
     ((null args) T)
     ((first args) (apply #'applicable-and (rest args)))
     (T NIL))))

USER: (applicable-and t 3 4 5 pi t)
T
USER: (applicable-and t 3 4 5 nil t)
NIL
```