- A lambda expression can use any variable that is in scope where the expression appears--e.g., *if it is in a function it can use the function's parameters*!
-
-

- A lambda expression can use any variable that is in scope where the expression appears--e.g., *if it is in a function it can use the function's parameters*!

- A ***closure*** is a function represented in the machine in such a way that, regardless of when and where the function is called, *the function has access to any variable that is in scope at the point where the function is defined or created.*

-

- A lambda expression can use any variable that is in scope where the expression appears--e.g., *if it is in a function it can use the function's parameters*!

- A ***closure*** is a function represented in the machine in such a way that, regardless of when and where the function is called, *the function has access to any variable that is in scope at the point where the function is defined or created*.

- When a function is passed as an argument into or returned as the result of a function call, the argument or result is a ***closure***.

- A lambda expression can use any variable that is in scope where the expression appears--e.g., *if it is in a function it can use the function's parameters*!

- A lambda expression can use any variable that is in scope where the expression appears--e.g., *if it is in a function it can use the function's parameters*!

Example from Touretzky:

6.6.4 SET-DIFFERENCE

The SET-DIFFERENCE function performs set subtraction. It returns what is left of the first set when the elements in the second set have been removed. Again, the order of elements in the result is undefined.

7.14. Here is a version of SET-DIFFERENCE written with REMOVE-IF:

- A lambda expression can use any variable that is in scope where the expression appears--e.g., *if it is in a function it can use the function's parameters*!

Example from Touretzky:

6.6.4 SET-DIFFERENCE

The SET-DIFFERENCE function performs set subtraction. It returns what is left of the first set when the elements in the second set have been removed. Again, the order of elements in the result is undefined.

```
> (set-difference '(alpha bravo charlie delta)
                  '(bravo charlie))
(ALPHA DELTA)

> (set-difference '(alpha bravo charlie delta)
                  '(echo alpha foxtrot))
(BRAVO CHARLIE DELTA)
```

7.14. Here is a version of SET-DIFFERENCE written with REMOVE-IF:

- A lambda expression can use any variable that is in scope where the expression appears--e.g., *if it is in a function it can use the function's parameters*!

Example from Touretzky:

6.6.4 SET-DIFFERENCE

The SET-DIFFERENCE function performs set subtraction. It returns what is left of the first set when the elements in the second set have been removed. Again, the order of elements in the result is undefined.

```
> (set-difference '(alpha bravo charlie delta)
                  '(bravo charlie))
(ALPHA DELTA)

> (set-difference '(alpha bravo charlie delta)
                  '(echo alpha foxtrot))
(BRAVO CHARLIE DELTA)
```

7.14. Here is a version of SET-DIFFERENCE written with REMOVE-IF:

```
(defun my-setdiff (x y)
  (remove-if #'(lambda (e) (member e y))
             x))
```

Having seen examples of how to *use* functions that take functions as arguments, we now consider:

Question 2: How do we *write* functions that take functions as arguments?

Question 2: How do we *write* functions that take functions as arguments?

Question 2: How do we *write* functions that take functions as arguments?

In Common Lisp the ***value*** of an identifier F (as a variable) and the ***function definition*** of F are two *unrelated and independent* attributes of F!

At any given time, an identifier F may have *neither*, *just one*, or *both* of these attributes.

Question 2: How do we *write* functions that take functions as arguments?

In Common Lisp the ***value*** of an identifier F (as a variable) and the ***function definition*** of F are two *unrelated and independent* attributes of F!

At any given time, an identifier F may have *neither*, *just one*, or *both* of these attributes.

In this regard Common Lisp is like Java (which allows, e.g., a class to have both a method F and a field F) and unlike C++.

Question 2: How do we *write* functions that take functions as arguments?

In Common Lisp the ***value*** of an identifier F (as a variable) and the ***function definition*** of F are two *unrelated and independent* attributes of F!

At any given time, an identifier F may have *neither*, *just one*, or *both* of these attributes.

In this regard Common Lisp is like Java (which allows, e.g., a class to have both a method F and a field F) and unlike C++.

(defun f ...) sets the ***function definition*** of F, but doesn't affect the value (if any) of F.

Question 2: How do we *write* functions that take functions as arguments?

In Common Lisp the ***value*** of an identifier F (as a variable) and the ***function definition*** of F are two *unrelated and independent* attributes of F!

At any given time, an identifier F may have *neither*, *just one*, or *both* of these attributes.

In this regard Common Lisp is like Java (which allows, e.g., a class to have both a method F and a field F) and unlike C++.

(defun f ...) sets the ***function definition*** of F, but doesn’t affect the value (if any) of F.

The following will set the ***value*** of F but won’t affect the function definition (if any) of F:

- (setf F ...)
- (let ((F ...) or (let* ((F ...)
- Parameter passing (if F is a formal parameter).

euclid.cs.qc.cuny.edu - PuTTY

```
Welcome to GNU CLISP 2.49 (2010-07-07) <http://clisp.cons.org/>

Copyright (c) Bruno Haible, Michael Stoll 1992, 1993
Copyright (c) Bruno Haible, Marcus Daniels 1994-1997
Copyright (c) Bruno Haible, Pierpaolo Bernardi, Sam Steingold 1998
Copyright (c) Bruno Haible, Sam Steingold 1999-2000
Copyright (c) Sam Steingold, Bruno Haible 2001-2010

Type :h and hit Enter for context help.

[1]> (defun f (x) (+ x 100000))
F
[2]> (setf f 111)
111
```

euclid.cs.qc.cuny.edu - PuTTY

```
Welcome to GNU CLISP 2.49 (2010-07-07) <http://clisp.cons.org/>

Copyright (c) Bruno Haible, Michael Stoll 1992, 1993
Copyright (c) Bruno Haible, Marcus Daniels 1994-1997
Copyright (c) Bruno Haible, Pierpaolo Bernardi, Sam Steingold 1998
Copyright (c) Bruno Haible, Sam Steingold 1999-2000
Copyright (c) Sam Steingold, Bruno Haible 2001-2010

Type :h and hit Enter for context help.

[1]> (defun f (x) (+ x 100000))
F
[2]> (setf f 111)
111
[3]> (f f)
100111
```

euclid.cs.qc.cuny.edu - PuTTY

```
Welcome to GNU CLISP 2.49 (2010-07-07) <http://clisp.cons.org/>

Copyright (c) Bruno Haible, Michael Stoll 1992, 1993
Copyright (c) Bruno Haible, Marcus Daniels 1994-1997
Copyright (c) Bruno Haible, Pierpaolo Bernardi, Sam Steingold 1998
Copyright (c) Bruno Haible, Sam Steingold 1999-2000
Copyright (c) Sam Steingold, Bruno Haible 2001-2010

Type :h and hit Enter for context help.

[1]> (defun f (x) (+ x 100000))
F
[2]> (setf f 111)
111
[3]> (f f)
100111
[4]> (let ((f 222))
       (f f))
100222
```

euclid.cs.qc.cuny.edu - PuTTY

```
Welcome to GNU CLISP 2.49 (2010-07-07) <http://clisp.cons.org/>

Copyright (c) Bruno Haible, Michael Stoll 1992, 1993
Copyright (c) Bruno Haible, Marcus Daniels 1994-1997
Copyright (c) Bruno Haible, Pierpaolo Bernardi, Sam Steingold 1998
Copyright (c) Bruno Haible, Sam Steingold 1999-2000
Copyright (c) Sam Steingold, Bruno Haible 2001-2010

Type :h and hit Enter for context help.

[1]> (defun f (x) (+ x 100000))
F
[2]> (setf f 111)
111
[3]> (f f)
100111
[4]> (let ((f 222))
       (f f))
100222
[5]> (defun g (f) (f f))
G
[6]> (g 333)
100333
[7]> 
```

Q: Can the ***value*** of a Lisp expression be a function?

A: Yes! Three important cases of this are:

Q: Can the ***value*** of a Lisp expression be a function?

A: Yes! Three important cases of this are:

- If G is a symbol that has a function definition, the ***value*** of #’G is G’s function definition.

Q: Can the ***value*** of a Lisp expression be a function?

A: Yes! Three important cases of this are:

- If G is a symbol that has a function definition, the ***value*** of #’G is G’s function definition.
- The ***value*** of a lambda expression is a function.

Q: Can the ***value*** of a Lisp expression be a function?

A: Yes! Three important cases of this are:

- If G is a symbol that has a function definition, the ***value*** of #'G is G's function definition.
- The ***value*** of a lambda expression is a function.
- You can make the ***value*** of a variable G a function using SETF, LET / LET*, or parameter passing.

Q: Can the ***value*** of a Lisp expression be a function?

A: Yes! Three important cases of this are:

- If G is a symbol that has a function definition, the ***value*** of #'G is G's function definition.
- The ***value*** of a lambda expression is a function.
- You can make the ***value*** of a variable G a function using SETF, LET / LET*, or parameter passing.

Q: When the ***value*** of a variable or other Lisp expression is a function, how can we *call* that function?

Q: Can the ***value*** of a Lisp expression be a function?

A: Yes! Three important cases of this are:

- If G is a symbol that has a function definition, the ***value*** of #’G is G’s function definition.
- The ***value*** of a lambda expression is a function.
- You can make the ***value*** of a variable G a function using SETF, LET / LET*, or parameter passing.

Q: When the ***value*** of a variable or other Lisp expression is a function, how can we *call* that function?

A: Use **FUNCALL** (or APPLY, which we’ll look at later).

Q: Can the ***value*** of a Lisp expression be a function?

A: Yes! Three important cases of this are:

- If G is a symbol that has a function definition, the ***value*** of #'G is G's function definition.
- The ***value*** of a lambda expression is a function.
- You can make the ***value*** of a variable G a function using SETF, LET / LET*, or parameter passing.

Q: When the ***value*** of a variable or other Lisp expression is a function, how can we *call* that function?

A: Use **FUNCALL** (or APPLY, which we'll look at later).

```
> (setf g #'(lambda (x) (* x 10)))
#<Lexical-closure 41653824>

> (funcall g 12)
120
```

From sec. 7.12 of Touretzky.

The value of the variable G is a lexical closure, which is a function. But G itself is not the name of any function; if we wrote (G 12) we would get an undefined function error.

euclid.cs.qc.cuny.edu - PuTTY

```
Welcome to GNU CLISP 2.49 (2010-07-07) <http://clisp.cons.org/>

Copyright (c) Bruno Haible, Michael Stoll 1992, 1993
Copyright (c) Bruno Haible, Marcus Daniels 1994-1997
Copyright (c) Bruno Haible, Pierpaolo Bernardi, Sam Steingold 1998
Copyright (c) Bruno Haible, Sam Steingold 1999-2000
Copyright (c) Sam Steingold, Bruno Haible 2001-2010

Type :h and hit Enter for context help.

[1]> (defun f (x) (+ x 10000))
F
[2]> (defun g (x) (+ x 500))
G
[3]> (setf g #'f)
#<FUNCTION F ...
```

Value of G is the function named by F.

euclid.cs.qc.cuny.edu - PuTTY

```
Welcome to GNU CLISP 2.49 (2010-07-07) <http://clisp.cons.org/>

Copyright (c) Bruno Haible, Michael Stoll 1992, 1993
Copyright (c) Bruno Haible, Marcus Daniels 1994-1997
Copyright (c) Bruno Haible, Pierpaolo Bernardi, Sam Steingold 1998
Copyright (c) Bruno Haible, Sam Steingold 1999-2000
Copyright (c) Sam Steingold, Bruno Haible 2001-2010

Type :h and hit Enter for context help.

[1]> (defun f (x) (+ x 10000))
F
[2]> (defun g (x) (+ x 500))
G
[3]> (setf g #'f)
#<FUNCTION F ...
[4]> (g 1)
501
```

Value of G is the function named by F.

(g 1) calls the function named by G.

euclid.cs.qc.cuny.edu - PuTTY

```
Welcome to GNU CLISP 2.49 (2010-07-07) <http://clisp.cons.org/>

Copyright (c) Bruno Haible, Michael Stoll 1992, 1993
Copyright (c) Bruno Haible, Marcus Daniels 1994-1997
Copyright (c) Bruno Haible, Pierpaolo Bernardi, Sam Steingold 1998
Copyright (c) Bruno Haible, Sam Steingold 1999-2000
Copyright (c) Sam Steingold, Bruno Haible 2001-2010

Type :h and hit Enter for context help.

[1]> (defun f (x) (+ x 10000))
F
[2]> (defun g (x) (+ x 500))
G
[3]> (setf g #'f)
#<FUNCTION F ...
[4]> (g 1)
501
[5]> (funcall g 1)
10001
```

Value of G is the function named by F.

(g 1) calls the function named by G.

(funcall g 1) calls the function given by the ***value*** of G.

euclid.cs.qc.cuny.edu - PuTTY

```
Welcome to GNU CLISP 2.49 (2010-07-07) <http://clisp.cons.org/>

Copyright (c) Bruno Haible, Michael Stoll 1992, 1993
Copyright (c) Bruno Haible, Marcus Daniels 1994-1997
Copyright (c) Bruno Haible, Pierpaolo Bernardi, Sam Steingold 1998
Copyright (c) Bruno Haible, Sam Steingold 1999-2000
Copyright (c) Sam Steingold, Bruno Haible 2001-2010

Type :h and hit Enter for context help.

[1]> (defun f (x) (+ x 10000))
F
[2]> (defun g (x) (+ x 500))
G
[3]> (setf g #'f)
#<FUNCTION F ...
[4]> (g 1)
501
[5]> (funcall g 1)
10001
[6]> (funcall #'g 1)
501
[7]>
```

Value of G is the function named by F.

(g 1) calls the function named by G.

(funcall g 1) calls the function given by the ***value*** of G.

(funcall #’g 1) calls the function given by the value of #’G, which is the function named by G.

FUNCALL Can Call Functions That Take 2 or More Args!

(funcall f e_1 ... e_k) calls the function given by the value of f; the values of e_1, ..., e_k are passed as arguments.

FUNCALL Can Call Functions That Take 2 or More Args!

(funcall f e_1 … e_k) calls the function given by the value of f; the values of e_1, ..., e_k are passed as arguments.

This example is from sec. 7.2 of Touretzky.

*It also serves as a reminder that if the 2nd arg of CONS isn't a list, CONS returns a **dotted** list!*

FUNCALL Can Call Functions That Take 2 or More Args!

(funcall f e_1 ... e_k) calls the function given by the value of f; the values of e_1, ..., e_k are passed as arguments.

```
> (setf fn #'cons)
#<Compiled-function CONS {6041410}>

> fn
#<Compiled-function CONS {6041410}>

> (type-of fn)
COMPILED-FUNCTION

> (funcall fn 'c 'd)
(C . D)
```

This example is from sec. 7.2 of Touretzky.

*It also serves as a reminder that if the 2nd arg of CONS isn't a list, CONS returns a **dotted** list!*

The value of the variable FN is a function object. TYPE-OF shows that the object is of type COMPILED-FUNCTION. So you see that functions and symbols are not the same. The symbol CONS serves as the name of the CONS function, but it is not the actual function. The relationship between functions and the symbols that name them is explained in Advanced Topics section 3.18.

Writing Functions That Take Functions as Arguments

- When computing a call of a function g, each formal parameter of g is a variable whose ***value*** is set to the corresponding actual argument.

Writing Functions That Take Functions as Arguments

- When computing a call of a function g, each formal parameter of g is a variable whose ***value*** is set to the corresponding actual argument.
- So if a formal parameter p corresponds to an actual argument that is a function, then we must use (**funcall** p ...) to call that function.

Writing Functions That Take Functions as Arguments

- When computing a call of a function g, each formal parameter of g is a variable whose ***value*** is set to the corresponding actual argument.

- So if a formal parameter p corresponds to an actual argument that is a function, then we must use (**funcall** p ...) to call that function.

  (p ...) and (funcall #'p ...) would ***not*** work here: They'd call *the function whose name is* p, rather than the function given by parameter p's value!

Example:

Writing Functions That Take Functions as Arguments

- When computing a call of a function g, each formal parameter of g is a variable whose ***value*** is set to the corresponding actual argument.

- So if a formal parameter p corresponds to an actual argument that is a function, then we must use **(funcall** p ...) to call that function.

  (p ...) and (funcall #'p ...) would ***not*** work here: They'd call *the function whose name is* p, rather than the function given by parameter p's value!

Example:

```
[1]> (defun yes-or-no (p x y)
        (if (funcall p x y)
            'yes
            'no))
YES-OR-NO
[2]> (yes-or-no #'> 30 29)
YES
[3]> (yes-or-no #'< 30 29)
NO
```

Writing Our Own Version of MAPCAR

```
(defun our-mapcar (f L)
  (if (endp L)
      nil
      (let ((X (our-mapcar f (cdr L))))
```

$\boxed{\text{an expression that computes (our-mapcar f L) from X and, possibly, f and/or L.}}$)))

Writing Our Own Version of MAPCAR

```
(defun our-mapcar (f L)
  (if (endp L)
      nil
      (let ((X (our-mapcar f (cdr L))))
```

an expression that computes `(our-mapcar f L)` from `X` and, possibly, `f` and/or `L`. `)))`

To write the expression above, recall:

```
  (MAPCAR #'sqrt '(9 4 1 16 0)) => (3 2 1 4 0)
  (MAPCAR #'sqrt   '(4 1 16 0)) =>   (2 1 4 0)
```

Thus:

Writing Our Own Version of MAPCAR

```
(defun our-mapcar (f L)
  (if (endp L)
      nil
      (let ((X (our-mapcar f (cdr L))))
```

an expression that computes (our-mapcar f L) from X and, possibly, f and/or L.)))

To write the $\boxed{}$ expression above, recall:

```
  (MAPCAR #'sqrt '(9 4 1 16 0)) => (3 2 1 4 0)
  (MAPCAR #'sqrt   '(4 1 16 0)) =>   (2 1 4 0)
```

Thus:

if L => (9 4 1 16 0) and f => $n \mapsto \sqrt{n}$

then X =>

and we want $\boxed{}$ **to** =>

Writing Our Own Version of MAPCAR

```
(defun our-mapcar (f L)
  (if (endp L)
      nil
      (let ((X (our-mapcar f (cdr L))))
           an expression that computes (our-mapcar f L)
           from X and, possibly, f and/or L.              )))
```

To write the $\boxed{}$ expression above, recall:

```
  (MAPCAR #'sqrt '(9 4 1 16 0)) => (3 2 1 4 0)
  (MAPCAR #'sqrt   '(4 1 16 0)) =>   (2 1 4 0)
```

Thus:

if L => (9 4 1 16 0) and f => $n \mapsto \sqrt{n}$

then X => (2 1 4 0)

and we want $\boxed{}$ **to** =>

Writing Our Own Version of MAPCAR

```
(defun our-mapcar (f L)
  (if (endp L)
      nil
      (let ((X (our-mapcar f (cdr L))))
```

an expression that computes (our-mapcar f L) from X and, possibly, f and/or L.)))

To write the [ ] expression above, recall:

```
  (MAPCAR #'sqrt '(9 4 1 16 0)) => (3 2 1 4 0)
  (MAPCAR #'sqrt   '(4 1 16 0)) =>   (2 1 4 0)
```

Thus:

if L => (9 4 1 16 0) and f => n $\mapsto \sqrt{n}$

then X => (2 1 4 0)

and we want [ ] **to** => (3 2 1 4 0)

=

Writing Our Own Version of MAPCAR

```
(defun our-mapcar (f L)
  (if (endp L)
      nil
      (let ((X (our-mapcar f (cdr L))))
```

> an expression that computes `(our-mapcar f L)` from `X` and, possibly, `f` and/or `L`.

```
)))
```

To write the $\boxed{}$ expression above, recall:

```
  (MAPCAR #'sqrt '(9 4 1 16 0)) => (3 2 1 4 0)
  (MAPCAR #'sqrt   '(4 1 16 0)) =>   (2 1 4 0)
```

Thus:

if `L => (9 4 1 16 0)` and `f =>` n $\mapsto \sqrt{n}$

then `X =>   (2 1  4 0)`

and we want $\boxed{}$ **to** `=> (3 2 1  4 0)`

`= (cons (funcall f (car L)) X)`

From this we see that

Writing Our Own Version of MAPCAR

```
(defun our-mapcar (f L)
  (if (endp L)
      nil
      (let ((X (our-mapcar f (cdr L))))
          an expression that computes (our-mapcar f L)
          from X and, possibly, f  and/or  L.                 )))
```

To write the $\boxed{\quad}$ expression above, recall:

```
  (MAPCAR #'sqrt '(9 4 1 16 0)) => (3 2 1 4 0)
  (MAPCAR #'sqrt   '(4 1 16 0)) =>   (2 1 4 0)
```

Thus:

if L => (9 4 1 16 0) and f => $n \mapsto \sqrt{n}$

then X => (2 1 4 0)

and we want $\boxed{\quad}$ **to** => (3 2 1 4 0)

= (cons (funcall f (car L)) X)

From this we see that

(cons (funcall f (car L)) X) is a good $\boxed{\quad}$ expression.

Writing Our Own Version of MAPCAR

```
(defun our-mapcar (f L)
  (if (endp L)
      nil
      (let ((X (our-mapcar f (cdr L))))
```

an expression that computes (our-mapcar f L) from X and, possibly, f and/or L.)))

From this we see that

(cons (funcall f (car L)) X) is a good $\boxed{}$ expression.

Writing Our Own Version of MAPCAR

```
(defun our-mapcar (f L)
  (if (endp L)
      nil
      (let ((X (our-mapcar f (cdr L))))
        (cons (funcall f (car L)) X)  )))
```

Writing Our Own Version of MAPCAR

```
(defun our-mapcar (f L)
  (if (endp L)
      nil
      (let ((X (our-mapcar f (cdr L))))
        (cons (funcall f (car L)) X)  )))
```

X isn’t used more than once, so we eliminate the LET:

```
(defun our-mapcar (f L)
  (if (endp L)
      nil
      ~~(let ((X (our-mapcar f (cdr L))))~~
     (cons (funcall f (car L)) (our-mapcar f (cdr L))) ~~)~~))
```

Writing the SIGMA Function

Recall that SIGMA should behave as follows:

```
If   g => a numerical function of one argument
and  j, k => integers,
then (sigma g j k) => g(j) + g(j+1) + ... + g(k).
                          [This sum is 0 if j > k.]
```

Q. What should we make smaller for the recursive call?

Writing the SIGMA Function

Recall that SIGMA should behave as follows:

```
  If   g => a numerical function of one argument
  and  j, k => integers,
  then (sigma g j k) => g(j) + g(j+1) + ... + g(k).
                           [This sum is 0 if j > k.]
```

Q. What should we make smaller for the recursive call?
A. The *no. of summands* (k-j+1 in the above example).

Writing the SIGMA Function

Recall that SIGMA should behave as follows:

```
If   g => a numerical function of one argument
and  j, k => integers,
then (sigma g j k) => g(j) + g(j+1) + ... + g(k).
                           [This sum is 0 if j > k.]
```

Q. What should we make smaller for the recursive call?
A. The *no. of summands* (k-j+1 in the above example).

```
(defun sigma (g j k)
  (if (> j k)
      0
      (let ((X (sigma g (+ j 1) k)))
           an expression that computes (sigma g j k)
           from X and, possibly, g, and/or j, and/or k   )))
```

Writing the SIGMA Function

```
(defun sigma (g j k)
  (if (> j k)
      0
      (let ((X (sigma g (+ j 1) k)))
        an expression that computes (sigma g j k)
        from X and, possibly, g, and/or j, and/or k   )))
```

Writing the SIGMA Function

```
(defun sigma (g j k)
  (if (> j k)
      0
      (let ((X (sigma g (+ j 1) k)))
            an expression that computes (sigma g j k)
            from X and, possibly, g, and/or j, and/or k  )))
```

Writing the SIGMA Function

```
(defun sigma (g j k)
  (if (> j k)
      0
      (let ((X (sigma g (+ j 1) k)))
```

$\boxed{\text{an expression that computes (sigma g j k) from X and, possibly, g, and/or j, and/or k}}$)))

To write the $\boxed{}$ expression above, consider:

(SIGMA (lambda (x) (* x x)) 3 6) => $3^2 + 4^2 + 5^2 + 6^2$

(SIGMA (lambda (x) (* x x)) 4 6) => $4^2 + 5^2 + 6^2$

Writing the SIGMA Function

```
(defun sigma (g j k)
  (if (> j k)
      0
      (let ((X (sigma g (+ j 1) k)))
```

an expression that computes (sigma g j k) from X and, possibly, g, and/or j, and/or k)))

To write the expression above, consider:

```
(SIGMA (lambda (x) (* x x)) 3 6) => 3² + 4² + 5² + 6²
(SIGMA (lambda (x) (* x x)) 4 6) =>      4² + 5² + 6²
```

Thus:

if g => $x \mapsto x^2$, j => 3, and k => 6

then X =>

and we want **to** =>

Writing the SIGMA Function

```
(defun sigma (g j k)
  (if (> j k)
      0
      (let ((X (sigma g (+ j 1) k)))
```

an expression that computes `(sigma g j k)` from `X` and, possibly, `g`, and/or `j`, and/or `k` `)))`

To write the $\boxed{}$ expression above, consider:

```
  (SIGMA (lambda (x) (* x x)) 3 6) => 3² + 4² + 5² + 6²
  (SIGMA (lambda (x) (* x x)) 4 6) =>      4² + 5² + 6²
```

Thus:

if g => $x \mapsto x^2$, j => 3, and k => 6

then X => $4^2 + 5^2 + 6^2$

and we want $\boxed{}$ **to** =>

Writing the SIGMA Function

```
(defun sigma (g j k)
  (if (> j k)
      0
      (let ((X (sigma g (+ j 1) k)))
           an expression that computes (sigma g j k)
           from X and, possibly, g, and/or j, and/or k  )))
```

To write the expression above, consider:

`(SIGMA (lambda (x) (* x x)) 3 6)` => $3^2 + 4^2 + 5^2 + 6^2$

`(SIGMA (lambda (x) (* x x)) 4 6)` => $4^2 + 5^2 + 6^2$

Thus:

if g => $x \mapsto x^2$, j => 3, and k => 6

then X => $4^2 + 5^2 + 6^2$

and we want **to** => $3^2 + 4^2 + 5^2 + 6^2$

=

Writing the SIGMA Function

```
(defun sigma (g j k)
  (if (> j k)
      0
      (let ((X (sigma g (+ j 1) k)))
           [an expression that computes (sigma g j k)
            from X and, possibly, g, and/or j, and/or k] )))
```

To write the $\boxed{}$ expression above, consider:

(SIGMA (lambda (x) (* x x)) 3 6) => $3^2 + 4^2 + 5^2 + 6^2$

(SIGMA (lambda (x) (* x x)) 4 6) => $4^2 + 5^2 + 6^2$

Thus:

if g => $x \mapsto x^2$, j => 3, and k => 6

then X => $4^2 + 5^2 + 6^2$

and we want $\boxed{}$ **to** => $3^2 + 4^2 + 5^2 + 6^2$

= (+ (funcall g j) X)

Writing the SIGMA Function

```
(defun sigma (g j k)
  (if (> j k)
      0
      (let ((X (sigma g (+ j 1) k)))
           an expression that computes (sigma g j k)
           from X and, possibly, g, and/or j, and/or k   )))
```

To write the $\boxed{\quad\quad}$ expression above, consider:

(SIGMA (lambda (x) (* x x)) 3 6) => $3^2 + 4^2 + 5^2 + 6^2$

(SIGMA (lambda (x) (* x x)) 4 6) => $4^2 + 5^2 + 6^2$

Thus:

if g => $x \mapsto x^2$, j => 3, and k => 6

then X => $4^2 + 5^2 + 6^2$

and we want $\boxed{\quad\quad}$ **to** => $3^2 + 4^2 + 5^2 + 6^2$

= (+ (funcall g j) X)

From this we see that

(+ (funcall g j) X) is a good $\boxed{\quad\quad}$ expression.

Writing the SIGMA Function

```
(defun sigma (g j k)
  (if (> j k)
      0
      (let ((X (sigma g (+ j 1) k)))
```

an expression that computes `(sigma g j k)` from `X` and, possibly, `g`, and/or `j`, and/or `k` `)))`

From this we see that
`(+ (funcall g j) X)` is a good expression.

Writing the SIGMA Function

```
(defun sigma (g j k)
  (if (> j k)
      0
      (let ((X (sigma g (+ j 1) k)))
        (+ (funcall g j) X) )))
```

Writing the SIGMA Function

```
(defun sigma (g j k)
  (if (> j k)
      0
      (let ((X (sigma g (+ j 1) k)))
        (+ (funcall g j) X) )))
```

X isn’t used more than once, so we eliminate the LET:

```
(defun sigma (g j k)
  (if (> j k)
      0
      (let ((X (sigma g (+ j 1) k)))
      (+ (funcall g j) (sigma g (+ j 1) k)) )))
```

MAPCAR Can Also Map Functions of Two or More Arguments

From Touretzky’s book

7.11 OPERATING ON MULTIPLE LISTS

In the beginning of this chapter we used MAPCAR to apply a one-input function to the elements of a list. MAPCAR is not restricted to one-input functions, however. Given a function of *n* inputs, MAPCAR will map it over *n* lists.

MAPCAR Can Also Map Functions of Two or More Arguments

7.11 OPERATING ON MULTIPLE LISTS

From Touretzky's book

In the beginning of this chapter we used MAPCAR to apply a one-input function to the elements of a list. MAPCAR is not restricted to one-input functions, however. Given a function of *n* inputs, MAPCAR will map it over *n* lists. For example, given a list of people and a list of jobs, we can use MAPCAR with a two-input function to pair each person with a job:

```
> (mapcar #'(lambda (x y) (list x 'gets y))
          '(fred wilma george diane)
          '(job1 job2 job3 job4))
((FRED GETS JOB1)
 (WILMA GETS JOB2)
 (GEORGE GETS JOB3)
 (DIANE GETS JOB4))
```

MAPCAR goes through the two lists in parallel, taking one element from each at each step.

MAPCAR Can Also Map Functions of Two or More Arguments

7.11 OPERATING ON MULTIPLE LISTS

From Touretzky's book

In the beginning of this chapter we used MAPCAR to apply a one-input function to the elements of a list. MAPCAR is not restricted to one-input functions, however. Given a function of *n* inputs, MAPCAR will map it over *n* lists. For example, given a list of people and a list of jobs, we can use MAPCAR with a two-input function to pair each person with a job:

```
> (mapcar #'(lambda (x y) (list x 'gets y))
          '(fred wilma george diane)
          '(job1 job2 job3 job4))
((FRED GETS JOB1)
 (WILMA GETS JOB2)
 (GEORGE GETS JOB3)
 (DIANE GETS JOB4))
```

MAPCAR goes through the two lists in parallel, taking one element from each at each step. If one list is shorter than the other, MAPCAR stops when it reaches the end of the shortest list.

MAPCAR Can Also Map Functions of Two or More Arguments

7.11 OPERATING ON MULTIPLE LISTS

From Touretzky's book

In the beginning of this chapter we used MAPCAR to apply a one-input function to the elements of a list. MAPCAR is not restricted to one-input functions, however. Given a function of *n* inputs, MAPCAR will map it over *n* lists. For example, given a list of people and a list of jobs, we can use MAPCAR with a two-input function to pair each person with a job:

```
> (mapcar #'(lambda (x y) (list x 'gets y))
          '(fred wilma george diane)
          '(job1 job2 job3 job4))
((FRED GETS JOB1)
 (WILMA GETS JOB2)
 (GEORGE GETS JOB3)
 (DIANE GETS JOB4))
```

MAPCAR goes through the two lists in parallel, taking one element from each at each step. If one list is shorter than the other, MAPCAR stops when it reaches the end of the shortest list.

Another example of operating on multiple lists is the problem of adding two lists of numbers pairwise:

```
> (mapcar #'+ '(1 2 3 4 5) '(60 70 80 90 100))
(61 72 83 94 105)

> (mapcar #'+ '(1 2 3) '(10 20 30 40 50))
(11 22 33)
```

Three More Examples of MAPCAR

```
[1]> (mapcar #'+ '(10000 20000 30000 40000)
                 '( 1000  2000  3000  4000)
                 '(  100   200   300   400)
                 '(   10    20    30    40)
                 '(    1     2     3     4))
(11111 22222 33333 44444)
[2]>
```

Three More Examples of MAPCAR

```
[1]> (mapcar #'+ '(10000 20000 30000 40000)
                 '( 1000  2000  3000  4000)
                 '(  100   200   300   400)
                 '(   10    20    30    40)
                 '(    1     2     3     4))
(11111 22222 33333 44444)
[2]>
===============================================================
[1]> (mapcar #'append '((A B)          (1 2 3 4)          (X))
                      '((P)            (Q R)              (S T U))
                      '((X X X X)      (Y Y Y)            (Z Z))
                      '((+ - *)        (/ %)              (^)))
((A B P X X X X + - *) (1 2 3 4 Q R Y Y Y / %) (X S T U Z Z ^))
[2]>
```

Three More Examples of MAPCAR

```
[1]> (mapcar #'+ '(10000 20000 30000 40000)
                 '( 1000  2000  3000  4000)
                 '(  100   200   300   400)
                 '(   10    20    30    40)
                 '(    1     2     3     4))
(11111 22222 33333 44444)
[2]>
==============================================================
[1]> (mapcar #'append '((A B)        (1 2 3 4)         (X))
                      '((P)          (Q R)             (S T U))
                      '((X X X X)    (Y Y Y)           (Z Z))
                      '((+ - *)      (/ %)             (^)))
((A B P X X X X + - *) (1 2 3 4 Q R Y Y Y / %) (X S T U Z Z ^))
[2]>

==============================================================
[1]> (mapcar (lambda (x y z) (list '1st x '2nd y '3rd z))
             '(ORANGE PLUM APPLE PEAR)
             '(ALPHA  BETA GAMMA DELTA)
             '(MOUSE  CAT  RAT   DOG))
((1ST ORANGE 2ND ALPHA 3RD MOUSE) (1ST PLUM 2ND BETA 3RD CAT)
 (1ST APPLE 2ND GAMMA 3RD RAT) (1ST PEAR 2ND DELTA 3RD DOG))
[2]>
```

Using APPLY to Apply a Function to a List of Arguments

APPLY is also a Lisp primitive function. APPLY takes a function and a list of objects as input. It invokes the specified function with those objects as its inputs.

From sec. 3.21 of Touretzky

Using APPLY to Apply a Function to a List of Arguments

APPLY is also a Lisp primitive function. APPLY takes a function and a list of objects as input. It invokes the specified function with those objects as its inputs.

From sec. 3.21 of Touretzky

```
(apply #'+ '(2 3)) ⇒ 5

(apply #'equal '(12 17)) ⇒ nil
```

Using APPLY to Apply a Function to a List of Arguments

APPLY is also a Lisp primitive function. APPLY takes a function and a list of objects as input. It invokes the specified function with those objects as its inputs.

From sec. 3.21 of Touretzky

```
(apply #'+ '(2 3)) ⇒ 5

(apply #'equal '(12 17)) ⇒ nil
```

The objects APPLY passes to the function are *not* evaluated first. In the following example, the objects are a symbol and a list. Evaluating either the symbol AS or the list (YOU LIKE IT) would cause an error.

```
(apply #'cons '(as (you like it)))
  ⇒ (as you like it)
```

Using APPLY to Apply a Function to a List of Arguments

APPLY is also a Lisp primitive function. APPLY takes a function and a list of objects as input. It invokes the specified function with those objects as its inputs.

From sec. 3.21 of Touretzky

```
(apply #'+ '(2 3)) ⇒ 5

(apply #'equal '(12 17)) ⇒ nil
```

The objects APPLY passes to the function are *not* evaluated first. In the following example, the objects are a symbol and a list. Evaluating either the symbol AS or the list (YOU LIKE IT) would cause an error.

```
(apply #'cons '(as (you like it)))
  ⇒ (as you like it)
```

Example: Use APPLY to write the SUM function

A. SUM is a function that is already defined on venus and euclid; if L *is any list of numbers* then (SUM L) returns the sum of the elements of L. [Thus (SUM ( )) returns 0.] Complete the following definition of a

of Lisp Assignment 4 without using recursion.

Solution:

Using APPLY to Apply a Function to a List of Arguments

From sec. 3.21 of Touretzky

APPLY is also a Lisp primitive function. APPLY takes a function and a list of objects as input. It invokes the specified function with those objects as its inputs.

```
(apply #'+ '(2 3)) ⇒ 5

(apply #'equal '(12 17)) ⇒ nil
```

The objects APPLY passes to the function are *not* evaluated first. In the following example, the objects are a symbol and a list. Evaluating either the symbol AS or the list (YOU LIKE IT) would cause an error.

```
(apply #'cons '(as (you like it)))
  ⇒ (as you like it)
```

Example: Use APPLY to write the SUM function

A. SUM is a function that is already defined on venus and euclid; if L *is any list of numbers* then (SUM L) returns the sum of the elements of L. [Thus (SUM ( )) returns 0.] Complete the following definition of a

of Lisp Assignment 4 without using recursion.

Solution: `(defun sum (L) (apply #'+ L))`

If f => a function and L => a list, then

(APPLY f $e_1 \ldots$ e_n L)

is evaluated by calling the function with the values of $e_1, \ldots, e_n$ as the first n arguments and the elements of the list as the remaining arguments.

The result returned by the function is returned by APPLY as its own result.

If f => a function and L => a list, then

(APPLY f e_1 … e_n L)

is evaluated by calling the function with the values of $e_1, \ldots, e_n$ as the first n arguments and the elements of the list as the remaining arguments.

The result returned by the function is returned by APPLY as its own result.

```
   (apply #'+ 2 20 200 2000 nil)
 = (apply #'+ 2 20 200 '(2000))
 = (apply #'+ 2 20 '(200 2000))
 = (apply #'+ 2 '(20 200 2000))
 = (apply #'+ '(2 20 200 2000)) = (+ 2 20 200 2000) => 2222
```

If f => a function and L => a list, then

(APPLY f e_1 ... e_n L)

is evaluated by calling the function with the values of $e_1, \ldots, e_n$ as the first n arguments and the elements of the list as the remaining arguments.

The result returned by the function is returned by APPLY as its own result.

```
   (apply #'+ 2 20 200 2000 nil)
 = (apply #'+ 2 20 200 '(2000))
 = (apply #'+ 2 20 '(200 2000))
 = (apply #'+ 2 '(20 200 2000))
 = (apply #'+ '(2 20 200 2000)) = (+ 2 20 200 2000) => 2222

   (apply #'mapcar #'+ '((1 2 3) (10 20 30) (100 200 300)))
 =
```

If f => a function and L => a list, then

(APPLY f e_1 ... e_n L)

is evaluated by calling the function with the values of $e_1, \ldots, e_n$ as the first n arguments and the elements of the list as the remaining arguments.

The result returned by the function is returned by APPLY as its own result.

```
   (apply #'+ 2 20 200 2000 nil)
 = (apply #'+ 2 20 200 '(2000))
 = (apply #'+ 2 20 '(200 2000))
 = (apply #'+ 2 '(20 200 2000))
 = (apply #'+ '(2 20 200 2000)) = (+ 2 20 200 2000) => 2222

   (apply #'mapcar #'+ '((1 2 3) (10 20 30) (100 200 300)))
 = (mapcar #'+  '(1   2   3)
               '(10  20  30)
              '(100 200 300))

            =>
```

If f => a function and L => a list, then

$$\text{(APPLY } f\ e_1 \ldots\ e_n\ L\text{)}$$

is evaluated by calling the function with the values of $e_1, \ldots, e_n$ as the first n arguments and the elements of the list as the remaining arguments.

The result returned by the function is returned by APPLY as its own result.

```
   (apply #'+ 2 20 200 2000 nil)
 = (apply #'+ 2 20 200 '(2000))
 = (apply #'+ 2 20 '(200 2000))
 = (apply #'+ 2 '(20 200 2000))
 = (apply #'+ '(2 20 200 2000)) = (+ 2 20 200 2000) => 2222

   (apply #'mapcar #'+ '((1 2 3) (10 20 30) (100 200 300)))
 = (mapcar #'+  '(1   2   3)
                '(10  20  30)
               '(100 200 300))

             => (111 222 333)
```

This example is relevant to problem 16(c) of Lisp Assignment 5!

Tail Recursive Functions and Tail Recursion Optimization

A call of a function F is said to be a ***tail call*** if the calling function ***does no additional work*** after that call of F returns--i.e.,

A call of a function F is said to be a ***tail call*** if the calling function ***does no additional work*** after that call of F returns--i.e., the calling function immediately returns control to its own caller, returning as ***its own*** result any value that was returned by the call of F.

For example,

A call of a function F is said to be a ***tail call*** if the calling function ***does no additional work*** after that call of F returns--i.e., the calling function immediately returns control to its own caller, returning as ***its own*** result any value that was returned by the call of F.

For example, in the following function definition

From p. 259 of Touretzky

```
(defun extract-symbols (x)
  (cond ((null x) nil)
        ((symbolp (first x))
         (cons (first x)
               (extract-symbols (rest x))))
        (t (extract-symbols (rest x)))))
```

the call of `cons` ***is a tail call***, and
the ***2nd*** recursive call of `extract-symbols` ***is a tail call***.

A call of a function F is said to be a ***tail call*** if the calling function ***does no additional work*** after that call of F returns--i.e., the calling function immediately returns control to its own caller, returning as ***its own*** result any value that was returned by the call of F.

For example, in the following function definition

From p. 259 of Touretzky

```
(defun extract-symbols (x)
  (cond ((null x) nil)
        ((symbolp (first x))
         (cons (first x)
               (extract-symbols (rest x))))
        (t (extract-symbols (rest x)))))
```

the call of `cons` ***is a tail call***, and
the ***2nd*** recursive call of `extract-symbols` ***is a tail call***.

But the calls of `null`, `symbolp`, `first`, and `rest`, and the ***1st*** recursive call of `extract-symbols` ***are not tail calls***!

A recursive call that is also a tail call is said to be a ***tail recursive*** call.

A recursive function is said to be ***tail recursive*** if ***every*** recursive call it makes is a tail call.

A recursive call that is also a tail call is said to be a ***tail recursive*** call.

A recursive function is said to be ***tail recursive*** if ***every*** recursive call it makes is a tail call.

Touretzky says this about such functions on p. G-14:

> **tail recursive**
>
> A function is tail recursive if it does all its work before making ~~the~~ each recursive call. Tail recursive functions return the result of ~~the~~ each recursive call without augmenting (modifying) it, or doing any other additional work. Clever Lisp compilers turn tail recursive calls into jump instructions, eliminating the need for a call stack. TYK

A recursive call that is also a tail call is said to be a ***tail recursive*** call.

A recursive function is said to be ***tail recursive*** if ***every*** recursive call it makes is a tail call.

Touretzky says this about such functions on p. G-14:

> **tail recursive**
>
> A function is tail recursive if it does all its work before making ~~the~~ each recursive call. Tail recursive functions return the result of ~~the~~ each recursive call without augmenting (modifying) it, or doing any other additional work. Clever Lisp compilers turn tail recursive calls into jump instructions, eliminating the need for a call stack.

TYK

The above function

From p. 259 of Touretzky

```
(defun extract-symbols (x)
  (cond ((null x) nil)
        ((symbolp (first x))
         (cons (first x)
               (extract-symbols (rest x))))
        (t (extract-symbols (rest x)))))
```

is ***not*** a tail recursive function, because only one of its two recursive calls is a tail recursive call.

The Concept of Tail Recursion is *Not* Specific to Lisp or Functional Programming

Java Examples:

```
static long f(int n, long r) // Returns n!*r when n >= 0
{
  if (n > 1) return f(n-1, n*r);    // A tail recursive call.
  else return r;
} Comment: This function works because  n! * r = (n-1)! * n*r
```

The Concept of Tail Recursion is *Not* Specific to Lisp or Functional Programming

Java Examples:

```
static long f(int n, long r) // Returns n!*r when n >= 0
{
  if (n > 1) return f(n-1, n*r);   // A tail recursive call.
  else return r;
} Comment: This function works because  n! * r = (n-1)! * n*r
```

An example of Tail Recursion in *Imperative* Programming:

```
static void reverseArray(int[] A, int i, int j)
// Reverses the subarray A[i .. j] of the array A[].
{
  if (i < j) {
    swap(A, i, j);                // Swap values in A[i] and A[j].
    reverseArray(A, i+1, j-1); // A tail recursive call
  }
}
```

Examples of Recursive Calls That are NOT Tail Recursive:

```
static void reverseArray(int[] A, int i, int j)
// Reverses the subarray A[i .. j] of the array A[].
{
  if (i < j) {
    reverseArray(A, i+1, j-1); // NOT a tail recursive call:
    swap(A, i, j);             // Swap is performed after call.
  }
}
```

Examples of Recursive Calls That are NOT Tail Recursive:

```
static void reverseArray(int[] A, int i, int j)
// Reverses the subarray A[i .. j] of the array A[].
{
  if (i < j) {
    reverseArray(A, i+1, j-1); // NOT a tail recursive call:
    swap(A, i, j);             // Swap is performed after call.
  }
}


static long f(int n) // returns n! when n >= 0
{
  if (n > 1) return n * f(n-1); // NOT a tail recursive call:
                                // * is performed after call
  else return 1;
}
```

A Compiler May Do *Tail Recursion Elimination**

*also called ***tail recursion optimization***

This replaces each ***tail recursive*** call of F with code that:

1.

2.

A Compiler May Do *Tail Recursion Elimination**

*also called ***tail recursion optimization***

This replaces each ***tail recursive*** call of F with code that:

1. Sets ***each formal parameter*** of F to the value of the corresponding actual argument of the tail recursive call.
2.

A Compiler May Do *Tail Recursion Elimination**

*also called ***tail recursion optimization***

This replaces each ***tail recursive*** call of F with code that:

1. Sets ***each formal parameter*** of F to the value of the corresponding actual argument of the tail recursive call.
2. Executes a jump that transfers control to the start of the body of the function F.

A Compiler May Do *Tail Recursion Elimination**

*also called ***tail recursion optimization***

This replaces each ***tail recursive*** call of F with code that:

1. Sets ***each formal parameter*** of F to the value of the corresponding actual argument of the tail recursive call.
2. Executes a jump that transfers control to the start of the body of the function F.

C++ Example to Illustrate this Transformation:

```
long f(int n, long r) {
  if (n > 1) {
    return f(n-1, n*r);
  }
  else return r;
}
```

⇒

A Compiler May Do *Tail Recursion Elimination**

*also called ***tail recursion optimization***

This replaces each ***tail recursive*** call of F with code that:

1. Sets ***each formal parameter*** of F to the value of the corresponding actual argument of the tail recursive call.
2. Executes a jump that transfers control to the start of the body of the function F.

C++ Example to Illustrate this Transformation:

```
long f(int n, long r) {
  if (n > 1) {
    return f(n-1, n*r);
  }
  else return r;
}
```

⇒

```
long f(int n, long r) {
 START: if (n > 1) {
          r = n*r;
          n = n-1;
          goto START;
        }
        else return r;
}
```

A Compiler May Do *Tail Recursion Elimination**

*also called ***tail recursion optimization***

This replaces each ***tail recursive*** call of F with code that:

1. Sets ***each formal parameter*** of F to the value of the corresponding actual argument of the tail recursive call.
2. Executes a jump that transfers control to the start of the body of the function F.

C++ Example to Illustrate this Transformation:

```
long f(int n, long r) {
  if (n > 1) {
    return f(n-1, n*r);
  }
  else return r;
}
```

⇒

```
long f(int n, long r) {
 START: if (n > 1) {
          r = n*r;
          n = n-1;
          goto START;
        }
        else return r;
}
```

Note: The code on the right updates r ***before*** updating n, as the new value of r (i.e. n*r) depends on n.

Tail Recursion Elimination May Be Necessary to Limit Memory Use When Depth of Recursion is High

The usual way to execute a function call written in a language that supports recursion involves ***allocating memory on a stack***

Tail Recursion Elimination May Be Necessary to Limit Memory Use When Depth of Recursion is High

The usual way to execute a function call written in a language that supports recursion involves ***allocating memory on a stack*** (to store values of parameters and other local variables of the called function, and to store the contents of the program counter and other registers that are in use at the time of the call when executing compiled code).

Tail Recursion Elimination May Be Necessary to Limit Memory Use When Depth of Recursion is High

The usual way to execute a function call written in a language that supports recursion involves ***allocating memory on a stack*** (to store values of parameters and other local variables of the called function, and to store the contents of the program counter and other registers that are in use at the time of the call when executing compiled code). **The allocated memory *will only be deallocated when the called function returns control to its caller*.**

Tail Recursion Elimination May Be Necessary to Limit Memory Use When Depth of Recursion is High

The usual way to execute a function call written in a language that supports recursion involves ***allocating memory on a stack*** (to store values of parameters and other local variables of the called function, and to store the contents of the program counter and other registers that are in use at the time of the call when executing compiled code). **The allocated memory *will only be deallocated when the called function returns control to its caller*.**

- This results in ***stack overflow*** if the depth of recursion is too great!

Tail Recursion Elimination May Be Necessary to Limit Memory Use When Depth of Recursion is High

The usual way to execute a function call written in a language that supports recursion involves ***allocating memory on a stack*** (to store values of parameters and other local variables of the called function, and to store the contents of the program counter and other registers that are in use at the time of the call when executing compiled code). **The allocated memory *will only be deallocated when the called function returns control to its caller*.**

- This results in ***stack overflow*** if the depth of recursion is too great!

The Clisp ***compiler*** does ***tail recursion elimination***, which ***eliminates the need to allocate memory for tail recursive calls***:

Tail Recursion Elimination May Be Necessary to Limit Memory Use When Depth of Recursion is High

The usual way to execute a function call written in a language that supports recursion involves ***allocating memory on a stack*** (to store values of parameters and other local variables of the called function, and to store the contents of the program counter and other registers that are in use at the time of the call when executing compiled code). **The allocated memory *will only be deallocated when the called function returns control to its caller*.**

- This results in ***stack overflow*** if the depth of recursion is too great!

The Clisp ***compiler*** does ***tail recursion elimination***, which ***eliminates the need to allocate memory for tail recursive calls***: So there's no limit on the depth of recursion for ***compiled*** tail recursive functions.

Example of Stack Overflow When Recursion is Too Deep

Write a function COUNTDOWN-FROM such that:
If n => a non-negative integer, then
(COUNTDOWN-FROM n) => a list of the integers from n down to 0.
Thus (countdown-from 10) => (10 9 8 7 6 5 4 3 2 1 0)

Example of Stack Overflow When Recursion is Too Deep

Write a function COUNTDOWN-FROM such that:

If n => a non-negative integer, then

(COUNTDOWN-FROM n) => a list of the integers from n down to 0.

Thus (countdown-from 10) => (10 9 8 7 6 5 4 3 2 1 0)

```
[1]> (defun countdown-from (n)
       (if (zerop n)
           '(0)
           (cons n (countdown-from (- n 1)))))
COUNTDOWN-FROM
```

Example of Stack Overflow When Recursion is Too Deep

Write a function COUNTDOWN-FROM such that:

If n => a non-negative integer, then

(COUNTDOWN-FROM n) => a list of the integers from n down to 0.

Thus (countdown-from 10) => (10 9 8 7 6 5 4 3 2 1 0)

```
[1]> (defun countdown-from (n)
       (if (zerop n)
           '(0)
           (cons n (countdown-from (- n 1)))))
COUNTDOWN-FROM
[2]> (countdown-from 10)
(10 9 8 7 6 5 4 3 2 1 0)
[3]> (length (countdown-from 20000))

*** - Program stack overflow. RESET
```

This depth of recursion is too great for the clisp interpreter!

Example of Stack Overflow When Recursion is Too Deep

Write a function COUNTDOWN-FROM such that:

If n => a non-negative integer, then
(COUNTDOWN-FROM n) => a list of the integers from n down to 0.

Thus (countdown-from 10) => (10 9 8 7 6 5 4 3 2 1 0)

```
[1]> (defun countdown-from (n)
       (if (zerop n)
           '(0)
           (cons n (countdown-from (- n 1)))))
COUNTDOWN-FROM
[2]> (countdown-from 10)
(10 9 8 7 6 5 4 3 2 1 0)
[3]> (length (countdown-from 20000))

*** - Program stack overflow. RESET
[4]> (compile 'countdown-from)
COUNTDOWN-FROM ;
NIL ;
NIL
[5]> (length (countdown-from 20000))
20001
[6]> (length (countdown-from 50000))
50001
```

This depth of recursion is too great for the clisp interpreter!

In addition to running faster, *compiled* code uses less stack stack space for function calls than interpreted code: So the recursion depth can be greater.

Example of Stack Overflow When Recursion is Too Deep

Write a function COUNTDOWN-FROM such that:

If n => a non-negative integer, then
(COUNTDOWN-FROM n) => a list of the integers from n down to 0.

Thus (countdown-from 10) => (10 9 8 7 6 5 4 3 2 1 0)

```
[1]> (defun countdown-from (n)
       (if (zerop n)
           '(0)
           (cons n (countdown-from (- n 1)))))
COUNTDOWN-FROM
[2]> (countdown-from 10)
(10 9 8 7 6 5 4 3 2 1 0)
[3]> (length (countdown-from 20000))

*** - Program stack overflow. RESET
[4]> (compile 'countdown-from)
COUNTDOWN-FROM ;
NIL ;
NIL
[5]> (length (countdown-from 20000))
20001
[6]> (length (countdown-from 50000))
50001
[7]> (length (countdown-from 100000))

*** - Program stack overflow. RESET
[8]>
```

This depth of recursion is too great for the clisp interpreter!

In addition to running faster, *compiled* code uses less stack stack space for function calls than interpreted code: So the recursion depth can be greater.

But recursion depth is still limited, because COUNTDOWN-FROM is ***not*** tail recursive!

```
[1]> (defun countdown-from-aux (hi lo accumulator)
       (if (< hi lo)
           accumulator
           (countdown-from-aux hi (+ lo 1) (cons lo accumulator))))
COUNTDOWN-FROM-AUX
```

This is a *tail recursive* helping function.

It returns the result of appending (hi ... lo) to the accumulator list.

```
[1]> (defun countdown-from-aux (hi lo accumulator)
       (if (< hi lo)
           accumulator
           (countdown-from-aux hi (+ lo 1) (cons lo accumulator))))
COUNTDOWN-FROM-AUX
[2]> (countdown-from-aux 50 43 '(the cat sat))
(50 49 48 47 46 45 44 43 THE CAT SAT)
```

This is a *tail recursive* helping function.

It returns the result of appending (hi ... lo) to the accumulator list.

```
[1]> (defun countdown-from-aux (hi lo accumulator)
       (if (< hi lo)
           accumulator
           (countdown-from-aux hi (+ lo 1) (cons lo accumulator))))
COUNTDOWN-FROM-AUX
[2]> (countdown-from-aux 50 43 '(the cat sat))
(50 49 48 47 46 45 44 43 THE CAT SAT)
[3]> (defun countdown-from (n) (countdown-from-aux n 0 nil))
COUNTDOWN-FROM
[4]> (countdown-from 10)
(10 9 8 7 6 5 4 3 2 1 0)
```

This is a *tail recursive* helping function.

It returns the result of appending (hi ... lo) to the accumulator list.

This definition of COUNTDOWN-FROM can take advantage of tail recursion elimination!

```
[1]> (defun countdown-from-aux (hi lo accumulator)
       (if (< hi lo)
           accumulator
           (countdown-from-aux hi (+ lo 1) (cons lo accumulator))))
COUNTDOWN-FROM-AUX
[2]> (countdown-from-aux 50 43 '(the cat sat))
(50 49 48 47 46 45 44 43 THE CAT SAT)
[3]> (defun countdown-from (n) (countdown-from-aux n 0 nil))
COUNTDOWN-FROM
[4]> (countdown-from 10)
(10 9 8 7 6 5 4 3 2 1 0)
[5]> (length (countdown-from 20000))

*** - Program stack overflow. RESET
```

This is a *tail recursive* helping function.

It returns the result of appending (hi ... lo) to the accumulator list.

This definition of COUNTDOWN-FROM can take advantage of tail recursion elimination!

<u>Before</u> COUNTDOWN-FROM-AUX is compiled, there's no tail recursion elimination and so recursion depth is limited!

```
[1]> (defun countdown-from-aux (hi lo accumulator)
       (if (< hi lo)
           accumulator
           (countdown-from-aux hi (+ lo 1) (cons lo accumulator))))
COUNTDOWN-FROM-AUX
[2]> (countdown-from-aux 50 43 '(the cat sat))
(50 49 48 47 46 45 44 43 THE CAT SAT)
[3]> (defun countdown-from (n) (countdown-from-aux n 0 nil))
COUNTDOWN-FROM
[4]> (countdown-from 10)
(10 9 8 7 6 5 4 3 2 1 0)
[5]> (length (countdown-from 20000))

*** - Program stack overflow. RESET
[6]> (compile 'countdown-from-aux)
COUNTDOWN-FROM-AUX ;
NIL ;
NIL
[7]> (length (countdown-from 20000))
20001
[8]> (length (countdown-from 200000))
200001
[9]> (length (countdown-from 2000000))
2000001
[10]> (length (countdown-from 20000000))
20000001
[11]>
```

This is a *tail recursive* helping function.

It returns the result of appending (hi ... lo) to the accumulator list.

This definition of COUNTDOWN-FROM can take advantage of tail recursion elimination!

Before COUNTDOWN-FROM-AUX is compiled, there's no tail recursion elimination and so recursion depth is limited!

After COUNTDOWN-FROM-AUX is compiled, **its recursion depth is no longer limited by the size of the stack**, as the compiler has performed ***tail recursion elimination***.

Differences Between Scheme and Common Lisp Regarding Functions That Take Functions as Arguments

Differences Between Scheme and Common Lisp Regarding Functions That Take Functions as Arguments

In Common Lisp, if F is the name of a certain function then the value of #’F is that same function, but F itself may have any value or no value. Moreover:

Differences Between Scheme and Common Lisp Regarding Functions That Take Functions as Arguments

In Common Lisp, if F is the name of a certain function then the value of #’F is that same function, but F itself may have any value or no value. Moreover:

- (DEFUN F ...) makes F the name of a function but does ***NOT*** affect the ***value*** (if any) of F.

Differences Between Scheme and Common Lisp Regarding Functions That Take Functions as Arguments

In Common Lisp, if F is the name of a certain function then the value of #’F is that same function, but F itself may have any value or no value. Moreover:

- (DEFUN F ...) makes F the name of a function but does ***NOT*** affect the ***value*** (if any) of F.
- If F is the name of a certain function, then (F ...) calls that function.

Differences Between Scheme and Common Lisp Regarding Functions That Take Functions as Arguments

In Common Lisp, if F is the name of a certain function then the value of #’F is that same function, but F itself may have any value or no value. Moreover:

- (DEFUN F ...) makes F the name of a function but does ***NOT*** affect the ***value*** (if any) of F.
- If F is the name of a certain function, then (F ...) calls that function.
- If the ***value*** of F is a certain function, then (FUNCALL F ...) calls that function.

Differences Between Scheme and Common Lisp Regarding Functions That Take Functions as Arguments

In Common Lisp, if F is the name of a certain function then the value of #’F is that same function, but F itself may have any value or no value. Moreover:

- (DEFUN F ...) makes F the name of a function but does ***NOT*** affect the ***value*** (if any) of F.
- If F is the name of a certain function, then (F ...) calls that function.
- If the ***value*** of F is a certain function, then (FUNCALL F ...) calls that function.

In Scheme, the ***value*** of F is a function ***just if*** F is the name of that same function. Accordingly:

Differences Between Scheme and Common Lisp Regarding Functions That Take Functions as Arguments

In Common Lisp, if F is the name of a certain function then the value of #'F is that same function, but F itself may have any value or no value. Moreover:

- (DEFUN F ...) makes F the name of a function but does ***NOT*** affect the ***value*** (if any) of F.
- If F is the name of a certain function, then (F ...) calls that function.
- If the ***value*** of F is a certain function, then (FUNCALL F ...) calls that function.

In Scheme, the ***value*** of F is a function ***just if*** F is the name of that same function. Accordingly:

- (FUNCALL F ...) is NOT used: A Common Lisp call (FUNCALL F ...) is written as (F ...) in Scheme.

Differences Between Scheme and Common Lisp Regarding Functions That Take Functions as Arguments

In Common Lisp, if F is the name of a certain function then the value of #’F is that same function, but F itself may have any value or no value. Moreover:

- (DEFUN F ...) makes F the name of a function but does ***NOT*** affect the ***value*** (if any) of F.
- If F is the name of a certain function, then (F ...) calls that function.
- If the ***value*** of F is a certain function, then (FUNCALL F ...) calls that function.

In Scheme, the ***value*** of F is a function ***just if*** F is the name of that same function. Accordingly:

- (FUNCALL F ...) is NOT used: A Common Lisp call (FUNCALL F ...) is written as (F ...) in Scheme.
- #’F is NOT used: A Scheme programmer would write F where a Common Lisp programmer writes #’F.