Functional Programming with Scheme

Characteristics of Imperative Languages:

- Principal operation is the assignment of values to variables.
- Programs are command oriented, and they carry out algorithms with command level sequence control, usually by selection and repetition.
- Computing is done by effect.

Problem Side effects in expressions.

Consequence Following properties are invalid in imperative languages:

Commutative, associative, and distributive laws for addition and multiplication

How can we reason about programs and their correctness if these fundamental properties of mathematics are fallacious?

Alternative: Functional Programming

Functional or applicative languages are based on the mathematical concept of a function.

- Concerned with data objects and values instead of variables.
- Principal operation is function application.
- Functions are treated as first-class objects that may be stored in data structures, passed as parameters, and returned as function results.
- Primitive functions are supplied, and the programmer defines new functions using functional forms.
- Program execution consists of the evaluation of an expression, and sequence control is by recursion.
- No assignment command; values communicated through the use of parameters.

- A discipline is enforced by functional languages:

    Side effects are avoided.

    The entire computation is summarized by the function value.

Principle of Referential Transparency

The value of a function is determined by the values of its arguments and the context in which the function application appears, and is independent of the history of the execution.

The evaluation of a function with the same argument produces the same value every time that it is invoked.

Lisp

Work on Lisp (List Processing) started in 1956 with an AI group at MIT under John McCarthy.

Principal versions are based on Lisp 1.5:

Common Lisp and Scheme

Features of Lisp

- High-level notation for lists.
- Recursive functions are emphasized.
- A program consists of a set of function definitions followed by a list of function evaluations.
- Functions are defined as expressions.
- Parameters are passed by value.

Scheme Syntax

Atoms

<atom> ::= <literal atom> | <numeric atom>

<literal atom> ::= <letter>
| <literal atom> <letter>
| <literal atom> <digit>

<numeric atom> ::= <numeral> | – <numeral>

<numeral> ::= <digit> | <numeral> <digit>

Atoms are considered indivisible.

Literal atoms consist of a string of alphanumeric characters usually starting with a letter.

Most Lisp systems allow any special characters in literal atoms as long as they cannot be confused with numbers.

Also, most Lisp systems allow floating-point numeric atoms.

S-expressions

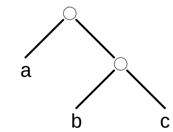
<S-expr> ::= <atom>
| **(** <S-expr> **.** <S-expr> **)**

"**(**", "**.**", and "**)**" are simply part of the syntactic representation of S-expressions—important feature is that an S-expr is a pair of S-exprs or an atom.

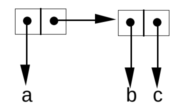
General S-expressions can be given a graphical representation:

(a . (b . c))

- **Lisp-tree** (or L-tree):



- **Cell-diagram** (or box notation):



Atoms have unique occurrences in S-expressions and can be shared.

Functions on S-expressions:

Selectors

car applied to a nonatomic S-expression, returns the left part.

cdr applied to a nonatomic S-expression, returns the right part.

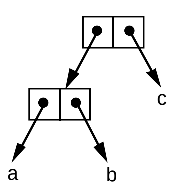
Examples

car[((a . b) . c)] = (a . b)

cdr[((a . b) . c)] = c

An error results if either is applied to an atom.

Implementation



car returns the left pointer.

cdr returns the right pointer.

A Constructor

cons applied to two S-expressions, returns the dotted pair containing them.

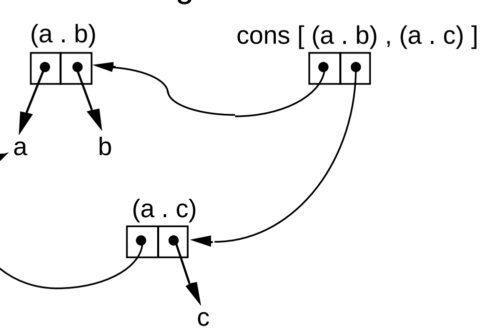
Examples

cons[p , q] = (p . q)

cons[(a . b) , (c . (a . d))] =
((a . b) . (c . (a . d)))

Implementation

Allocate a new cell and set its left and right pointers to the two arguments.



Lists

Notion of an S-expression is too general for most computing tasks, so Scheme deals primarily with a subset of the S-expressions: Lists.

Definition of Lists

1. The special atom ( ) is a list.

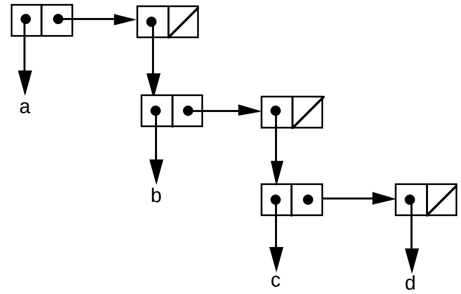
   ( ) is the only S-expression that is both an atom and a list; it denotes the empty list.

2. A dotted pair is a list if its right (cdr) element is a list.

S-expressions that are lists use special notation:

(a . ())	is represented by	(a)
(b . (a . ()))	is represented by	(b a)
(c . (b . (a . ())))	is represented by	(c b a)

Cell-diagrams for Lists



Functions on Lists

car	When applied to a nonempty list, returns the first element of the list.
cdr	When applied to a nonempty list, returns the list with the first element removed.
cons	When applied to an arbitrary S-expression and a list, returns the list obtained by appending the first argument onto the beginning of the list (the second argument).

Examples

car[(a b c)] = a   cdr[(a b c)] = (b c)

car[((a))] = (a)   cdr[((a))] = ()

cons[(a) , (b c)] = ((a) b c)

cons[a , ()] = (a)

Syntax for Functions

Application of a function to a set of arguments is expressed as a list:

(function-name sequence-of-arguments)

Notation is called **Cambridge Polish Form**

Predefined Numeric Functions

Unary functions

(add1 0)	returns	1
(add1 (abs -5))	returns	6
(sub1 -5)	returns	-6

Binary functions

(- 16 9)	returns	7
(quotient 17 5)	returns	3
(/ 17 5)	returns	3.4
(- (* 10 2) (+ 13 3))	returns	4

N-ary functions:

(+ 1 2 3 4 5)	returns	15
(* 1 2 3 4 5)	returns	120
(max 2 12 3 10)	returns	12
(min (* 4 6) (+ 4 6) (- 4 6))	returns	-2

Miscellaneous functions

(expt 2 5)	returns	32
(sqrt 25)	returns	5
(sqrt 2)	returns	1.4142135623730951
(sin 1)	returns	0.8414709848078965
(random 100)	returns	87, then 2, …

Predefined Predicate Functions

These are the Boolean functions of Scheme.

They return either the atom #t (for true) or the atom #f (for false).

(negative? -6)	returns #t
(zero? 44)	returns #f
(positive? -33)	returns #f
(number? 5)	returns #t
(integer? 3.7)	returns #f
(> 6 2)	returns #t
(= 6 2)	returns #f
(>= 3 30)	returns #f
(<= -5 -3)	returns #t
(odd? 5)	returns #t
(even? 37)	returns #f

Scheme Evaluation

When the Scheme interpreter encounters an atom, it evaluates the atom:

- Numeric atoms evaluate to themselves.
- Literal atoms #t and #f evaluate to themselves.
- All other literal atoms may have a value associated with them.

A value may be bound to an atom using the “define” operation, which makes the binding and returns a value:

(define a 5)	returns a
(define b 3)	returns b
a	returns 5
(+ a b)	returns 8
(+ a c)	returns ERROR

When the Scheme interpreter encounters a list, it expects the first item in the list to be an atom (or special operator) that represents a function.

The rest of the items in the list are evaluated and given to the function as argument values.

(* a (add1 b)) returns 20

Question

How does one apply car to the list (a b c)?

(car (a b c)) means that “a” is a function, applied to the values of “b” and “c”, whose value is passed to car.

Quote

Scheme evaluation is inhibited by the quote operation.

(quote a)	returns a unevaluated
(quote (a b c))	returns (a b c) unevaluated
(car (quote (a b c)))	returns a
(cdr (quote (a b c)))	returns (b c)
(cons (quote x) (quote (y z)))	returns list (x y z)

Quote may be abbreviated in the following way:

(cdr '((a) (b) (c))) returns ((b) (c))

(cons 'p '(q)) returns (p q)

Other Predefined Functions (Predicates)

pair? when applied to any S-expression, returns #t if it is a pair, #f otherwise.

(pair? 'x)	returns #f
(pair? '(x))	returns #t

atom? is the logical negation of pair? (not standard in Scheme)

null? when applied to any S-expression, returns #t if it is the empty list, #f otherwise.

(null? '( ))	returns #t
(null? '(( )))	returns #f

eq? when applied to two *atoms*, returns #t if they are equal, #f otherwise.

(eq? 'xy 'x)	returns #f

(eq? (pair? 'gonzo) #f)   returns #t

(eq? '(foo) '(foo))   returns #f

Abbreviations for car and cdr

(car (cdr (cdr '(a b c)))) may be
abbreviated (caddr '(a b c))

Problem with eq?

Expression (eq? x y) tests the equality of the values of x and y.

Given the bindings:

(define x '(a b)) and (define y '(a b)),

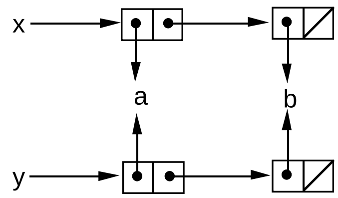
x   returns (a b), and

y   returns (a b), but

(eq? x y) returns #f

Although the values appear to be the same, they are two *different* copies of the same S-expression.

The test (eq? x y) returns #f because x and y point to two separate objects.



But (eq? (car x) (car y)) returns #t because (literal) atoms are always unique.

Special Forms

All the operations considered so far do not act in the same way.

True Scheme functions always evaluate their arguments.

When (+ (car '(2 4 6)) 5) is submitted to the interpreter, each item is evaluated:

+	evaluates to the predefined addition operation
(car '(2 4 6))	evaluates to the number 2
5	evaluates to the number 5.

Several of the operations described so far do not and cannot evaluate all of their operands.

(quote a)   simply returns its operand unevaluated.

(define x (+ 5 6)) evaluates its second argument, but leaves its first argument unevaluated.

These operations are called **special forms** to distinguish them from normal Scheme functions.

Complete list of special forms in Scheme

and	do	or
begin	if	quasiquote
case	lambda	quote
cond	let	set!
define	let*	while
delay	letrec	

Defining Functions in Scheme

Special form "define" returns the name of function being defined with the side effect of binding an expression defining a function to that name.

```
(define name
    (lambda (list-of-parameters) expression))
```

Examples:

```
(define disc (lambda (a b c)
        (sqrt (- (* b b)
                 (* 4 a c) )) ))
```

(disc 3 10 3) returns 8

(disc 5 8 -4)   returns 12

(define first (lambda (L) (car L)))

(define second (lambda (L) (car (cdr L))))

(first '((a b c)))   returns (a b c)

(second '((a) (b) (c))) returns (b)

Conditional Form

Decisions in Scheme are represented as conditional expressions using the special form **cond**

(cond (c_1 e_1) (c_2 e_2) … (c_n e_n) (else e_{n+1}))

which is equivalent to

if c_1 **then** return e_1
else if c_2 **then** return e_2
:
else if c_n **then** return e_n
else return e_{n+1}

If all of c_1, c_2, …, c_n are false and the else clause is omitted, then the cond result is unspecified.

Note that for the purposes of testing, any non-#f value represents true.

Each condition in a cond may be followed by a sequence of expressions whose last value is the result returned.

The other expressions are evaluated for their side effect only, say for output.

```
(cond ((= n 0) (display “zero”) 0)
      ((positive? n) (display ‘positive) 1)
      (else (display ‘negative) -1))
```

If

Another special form for decision making is the “if” operation.

(if test then-expression else-expression)

Example

```
(if (zero? n)
    0
    (/ m n))
```


Inductive or Recursive Definitions

Main control structure in Scheme is recursion.

Many functions can be defined inductively.

Example 1 Factorial

0! = 1
n! = n • (n-1)! for n>0

```
(define fact (lambda (n)
  (cond ((zero? n) 1))
        (else (* n (fact (sub1 n)))) )))
```

Sample execution:

(fact 4)
= 4 • (fact 3)
= 4 • [3 • (fact 2)]
= 4 • [3 • [2 • (fact 1)]]
= 4 • [3 • [2 • [1• (fact 0)]]]
= 4 • [3 • [2 • [1• 1]]] = 24

Example 2 GCD (assume a>0)

gcd(a,0) = a
gcd(a,b) = gcd(b,a **mod** b) if b>0

```
(define gcd (lambda (a b)
  (cond ((zero? b) a)
        (else (gcd b (modulo a b))) )))
```

Example 3 91-function:

F(n) = n - 10 if n>100
F(n) = F(F(n+11)) otherwise

```
(define F (lambda (n)
  (cond ((> n 100) (- n 10))
        (else (F (F (+ n 11))) )))
```

Lambda Notation

The anonymous function $\lambda x,y \,.\, y^2+x$ is represented in Scheme as

```
(lambda  (x y)  (+ (* y y) x)).
```

It can be used in a function application in the same way as a named function:

```
((lambda  (x y)  (+ (* y y) x))  3 4) returns 19.
```

When we define a function, we are simply binding a lambda expression to an identifier:

```
(define  fun  (lambda  (x y)  (+ (* y y) x)))
(fun  3 4)  returns  19.
```

Note that lambda is a special form.

Recursive Functions on Lists

1. Number of occurrences of atoms in a list of atoms:

   For example, (count1 '(a b c b a)) returns 5.

   Case 1 List is empty => return 0

   Case 2 List is not empty =>

   it has a first element that is an atom =>

   return 1 + number of atoms in cdr of list

```
(define  count1  (lambda (L)
    (cond  ((null? L)  0)
           (else  (add1 (count1 (cdr L)))) )))
```

2. Number of occurrences of atoms at the top level in an arbitrary list:

   (count2 '(a (b c) d a)) returns 3.

   Case 1 List is empty => return 0

   Case 2 List is not empty

   Subcase a First element is an atom =>

   return 1 + number of atoms in cdr of list

Subcase b First element is not an atom =>

return number of atoms in cdr of list.

```
(define  count2  (lambda (L)
  (cond  ((null? L)  0)
         ((atom? (car L))  (add1 (count2 (cdr L))))
         (else  (count2 (cdr L)))  )))
```

3. Number of occurrences of atoms at all levels in an arbitrary list:

   (count3 '(a (b c) b (a))) returns 5.

   Case 1 List is empty => return 0

   Case 2 List is not empty

   Subcase a First element is an atom =>

   return 1 + number of atoms in cdr of list

   Subcase b First element is not an atom =>

   return number of atoms in car of list

   + number of atoms in cdr of list

```
(define  count3  (lambda (L)
 (cond  ((null? L)  0)
        ((atom? (car L))  (add1 (count3 (cdr L))))
        (else (+ (count3 (car L)) (count3 (cdr L))))))
```


More Functions on Lists

Length of a list

```
(define  length  (lambda (L)
    (cond  ((null? L)  0)
           (else  (add1 (length (cdr L)))) )))
```

This function works the same as the predefined length function except for speed and storage.

Equality of arbitrary S-expressions

- Use = for numeric atoms
- Use eq? for literal atoms
- Otherwise, use recursion to compare left parts and right parts

```
(define  equal?  (lambda (s1 s2)
   (cond  ((number? s1)  (= s1 s2))
          ((atom? s1)  (eq? s1 s2))
          ((atom? s2)  #f)
          ((equal?  (car s1)  (car s2))
                    (equal?  (cdr s1) (cdr s2)))
          (else  #f)  )))
```

Concatenate two lists

```
(define concat (lambda (L1 L2)
  (cond ((null? L1) L2)
        (else (cons (car L1)
                    (concat (cdr L1) L2))))))
```

For example, (concat '(a b c) '(d e)) becomes

```
(cons 'a (concat '(b c) '(d e))) =
(cons 'a (cons 'b (concat '(c) '(d e)))) =
(cons 'a (cons 'b (cons 'c (concat '() '(d e))))) =
(cons 'a (cons 'b (cons 'c '(d e)))) = (a b c d e)
```

Reverse a list

```
(define reverse (lambda (L)
  (cond ((null? L) '())
        (else (concat (reverse (cdr L))
                      (list (car L)))) )))
```


An improved reverse

Use a help function and a collection variable.

```
(define rev (lambda (L) (help L '())))

(define help (lambda (L cv)
  (cond ((null? L) cv)
        (else (help (cdr L) (cons (car L) cv))) )))
```

Membership in a list (at the top level)

```
(define member (lambda (e L)
  (cond ((null? L) #f)
        ((equal? e (car L)) L)
        (else (member e (cdr L))) )))
```

This Boolean function returns the rest of the list starting with the matched element for true.

This behavior is consistent with the interpretation that any non-#f object represents true.

Logical operations

```
(define and (lambda (s1 s2)
        (cond (s1 s2) (else #f) )))
(define or (lambda (s1 s2)
        (cond (s1 s1) (s2 s2) (else #f) )))
```


Scope Rules in Scheme

In Lisp 1.5 and many of its successors access to nonlocal variables is resolved by **dynamic scoping** the calling chain is following until the variable is found local to a function.

Scheme and Common Lisp use **static scoping** nonlocal references are resolved at the point of function definition.

Static scoping is implemented by associating a closure (instruction pointer and environment pointer) with each function as it is defined.

The run-time execution stack maintains static links for nonlocal references.

Top-level define's create a global environment composed of the identifiers being defined.

A new scope is created in Scheme when the formal parameters, which are local variables, are bound to actual values when a function is invoked.

Local scope can be created by the let expression.

(let ((id_1 val_1) … (id_n val_n)) expr)

Expression (let ((a 5) (b 8)) (+ a b)) is an abbreviation of the function application

((lambda (a b) (+ a b)) 5 8);

Both expressions return the value 13.

Also has a sequential let, called let*, that evaluates the bindings from left to right.

(let* ((a 5) (b (+ a 3))) (* a b)) is equivalent to

(let ((a 5)) (let ((b (+ a 3))) (* a b))).

Finally, letrec must be used to bind an identifier to a function that calls the identifier recursively.

Define fact as an identifier local to the expression.

```
>>> (letrec ((fact (lambda (n)
                (cond ((zero? n) 1)
                      (else (* n (fact (sub1 n))))))))
       (fact 5))
120
```

Proving Correctness in Scheme

Correctness of programs in imperative languages is difficult to prove:

- Execution depends on the contents of each memory cell (each variable).
- Loops must be mentally executed.
- The progress of the computation is measured by snapshots of the state of the computer after every instruction.

Functional languages are much easier to reason about because of referential transparency: only those values immediately involved in a function application need be considered.

Programs defined as recursive functions usually can be proved correct by an induction proof.

Example

```
(define expr (lambda (a b)
        (if  (zero? b)
             1
             (if  (even? b)
                  (expr (* a a) (/ b 2))
                  (* a (expr a (sub1 b))) ))))
```

Precondition: $b \geq 0$

Postcondition: (expr a b) = a^b

Proof of correctness: By induction on b.

Basis: b = 0

Then $a^b = a^0 = 1$ and
(expr a b) = (expr a 0) returns 1.

Induction step: Suppose that for any c<b,
(expr a c) = a^c.

Let b>0 be an integer.

Case 1: b is even

Then
(expr a b) = (expr (* a a) (/ b 2))
= $(a \bullet a)^{b/2}$ by the induction hypothesis
= a^b

Case 2: b is odd (not even)

Then
(expr a b) = (* a (expr a (sub1 b)))
= $a \bullet (a^{b-1})$ by the induction hypothesis
= a^b

Higher-Order Functions

Expressiveness of functional programming comes from treating functions as first-class objects.

Scheme functions can be bound to identifiers using define and also be stored in structures:

```
(define  fn-list  (list add1  –  (lambda (n) (* n n))))
```

or alternatively

```
(define fn-list
  (cons add1 (cons – (cons (lambda (n) (* n n)) ‘()))))
```

defines a list of three unary functions.

fn-list returns (#<PROC add1> #<PROC –> #<PROC>)

Procedure to apply each function to a number:

```
(define construction
  (lambda (fl x)
    (cond  ((null? fl) '())
           (else (cons  ((car fl) x)
                        (construction (cdr fl) x))))))
```

so that

(construction fn-list 5) returns (6 -5 25)

Definition: A function is called **higher-order** if it has a function as a parameter or returns a function as its result.

Composition

```
(define compose
        (lambda (f g) (lambda (x) (f (g x)))))

(define inc-sqr
        (compose add1 (lambda (n) (* n n))))

(define sqr-inc
        (compose (lambda (n) (* n n)) add1))
```

Note that these two functions, inc-sqr and sqr-inc are defined without the use of parameters.

(inc-sqr 5) returns 26

(sqr-inc 5) returns 36

Apply to all

In Scheme "apply to all" is called map and is predefined, taking a unary function and a list as arguments, applying the function to each element of the list, and returning the list of results.

```
(map  add1  '(1 2 3))  returns  (2 3 4)

(map  (lambda (n) (* n n))  '(1 2 3))
                            returns  (1 4 9)

(map  (lambda (ls) (cons 'a ls))  '((b c) (a) ()))
                    returns  ((a b c) (a a) (a))
```

Map can be defined as follows:

```
(define map
  (lambda (proc lst)
    (if   (null? lst)
          '()
          (cons (proc (car lst)) (map proc (cdr lst))))))
```


Reduce or Accumulate

Higher-order functions are developed by abstracting common patterns from programs.

Consider the functions that find the sum or the product of a list of numbers:

```
(define sum
  (lambda (ls)
    (cond  ((null? ls) 0)
            (else (+ (car ls) (sum (cdr ls)))))))

(define product
  (lambda (ls)
    (cond    ((null? ls) 1)
      (else (* (car ls) (product (cdr ls)))))))
```

Common pattern:

```
(define reduce
  (lambda (proc init ls)
    (cond  ((null? ls) init)
      (else (proc     (car ls)
        (reduce proc init (cdr ls)))))))
```


Sum and product can be defined using reduce:

```
(define sum  (lambda (ls) (reduce + 0 ls)))

(define product  (lambda (ls) (reduce * 1 ls)))
```

Filter

By passing a Boolean function, filter in only those elements from a list that satisfy the predicate.

```
(define filter
  (lambda (proc ls)
     (cond ((null? ls)  '())
            ((proc (car ls)) (cons  (car ls)
                                    (filter proc (cdr ls))))
            (else (filter proc (cdr ls))) )))
```

```
(filter even? '(1 2 3 4 5 6))  returns  (2 4 6)

(filter (lambda (n) (> n 3)) '(1 2 3 4 5))  returns  (4 5)
```

Currying

A binary functions, for example, + or cons, takes both of its arguments at the same time.

(+ a b) will evaluate both a and b so that values can be passed to the addition operation.

It may be advantageous to have a binary function take its arguments one at a time.

Such a function is called **curried**

```
(define curried+
    (lambda (m)
        (lambda (n)  (+ m n)) ))
```

Note that if only one argument is supplied to curried+, the result is a function of one argument.

```
(curried+ 5)        returns  #<procedure>
((curried+ 5) 8)    returns  13
```

Unary functions can be defined using curried+:

```
(define add2  (curried+ 2))
(define add5  (curried+ 5))
```


Curried Map

```
(define cmap
  (lambda (proc)
    (lambda  (lst)
      (if  (null? lst)
           '()
           (cons  (proc (car lst))
                  ((cmap proc) (cdr lst))))))
```

```
(cmap add1)  returns  #<procedure>

((cmap add1)  '(1 2 3))  returns  (2 3 4)

((cmap  (cmap add1))  '((1) (2 3) (4 5 6)))
                        returns  ((2) (3 4) (5 6 7))

(((compose cmap cmap)  add1)  '((1) (2 3) (4 5 6)))
                        returns  ((2) (3 4) (5 6 7))
```

The notion of currying can be applied to functions with more than two arguments.

Tail Recursion

Functional programming is criticized for use of recursion and its inefficiency.

Scheme and some other functional languages have a mechanism whereby implementations optimize certain recursive functions by reducing the storage on the run-time execution stack.

Example Factorial

```
(define factorial
    (lambda (n)
        (if (zero? n)
            1
            (* n (factorial (sub1 n))) )))
```

When (factorial 6) is invoked, activation records are needed for seven invocations of the function, namely (factorial 6) through (factorial 0).

At its deepest level of recursion all the information in the expression,

(* 6 (* 5 (* 4 (* 3 (* 2 (* 1 (factorial 0))))))),

is stored in the run-time execution stack.

Example Factorial with Tail Recursion

```
(define fact
  (lambda (n)
     (letrec
        ((fact-help (lambda (prod count)
              (if (> count n)
                  prod
                  (fact-help  (* count prod)
                              (add1 count)) ))))
          (fact-help 1 1))))
```

No need to save local environment when recursive call made, since no computation remains.

Definition: A function is **tail recursive** if its only recursive call is the last action that occurs during any particular invocation of the function.

Execution of (fact 6) proceeds as follows:

```
(fact 6)
    (fact-help 1 1)
    (fact-help 1 2)
    (fact-help 2 3)
    (fact-help 6 4)
    (fact-help 24 5)
    (fact-help 120 6)
    (fact-help 720 7)
```