Recursion

Functional languages accomplish repeat action via recursion

```
%%  these factorials all assume N is a non-negative integer

fac(N) when N == 0 -> 1;
fac(N) when N > 0 -> N*fac(N-1).

fac2(0) -> 1;
fac2(N) when N > 0 -> N*fac2(N-1).

fac3(0) -> 1;
fac3(N) -> N*fac3(N-1).
```

Repeat Action Using Recursion

- How to “loop” in functional languages:

  1. Put all items to be processed into a list
  2. Write recursive function that shortens the list until it is empty

- Example:

```
list_len( [] ) -> 0;
list_len( [ _ | Tail ] ) -> 1 + list_len(Tail).
```

How to Loop

- Common for Erlang function to call itself as last action in body

- Here is a message receive “loop:”

```
rcv_loop() ->
   receive
      {Sender, Msg} ->
         io:fwrite("Received ~p from ~s~n", [Msg, Sender]),
         rcv_loop()
   end.
```

Erlang Processes

- In Erlang, “process” means neither process nor thread (more like a function call or “green threads”)

`www.erlang.org/doc/efficiency_guide/processes.html` says “A newly spawned Erlang process uses 309 words of memory ... The default initial heap size of 233 words is quite conservative in order to support Erlang systems with hundreds of thousands or even millions of processes.”

- Entire Erlang VM and all its “processes” execute as a single OS process

Thread Creation Reminder

- C thread creation:

  - `pthread_create` executes given function
  - Function takes single `void *` argument
  - Function returns `void *`

- Java thread creation:

  - Create class that implements `Runnable` or `Callable<T>` interface
  - Arguments passed to class constructor
  - Give `Runnable/Callable<T>` object to `Thread`, thread pool, or helper class such as `FutureTask`
  - `Runnable` returns no value
  - `Callable` returns value of type `T`; retrieve value from an associated object that implements `Future`

Erlang Process Creation

- `spawn/3`:

```
spawn( module_name, function_name, list_argument )
```

executes this:

```
module_name:function_name(list_argument)
```

in a separate Erlang "process"

- Erlang process creation:

- `spawn/3` executes given function
- Function takes list argument; list may have any number of items
- Function doesn't return a value
- `spawn` returns a Pid (process ID)

Process Communication, I

- Even though processes are part of same program, language provides no way for separate processes to access each other's variables

- Also: Erlang does not have global variables

Q: How do Erlang processes communicate?

A: Send messages to destination's **mailbox**

Q: What is a mailbox?

A: Queue of messages

Process Communication, II

- `spawn/3` returns a process ID (PID)
- To send a message:

```
PID ! expr
```

- `expr` is evaluated then the value is sent to process PID

Example

```
%%   spawn/3 creates new process that
%%   executes module:function([])
NewProcess = spawn(module, function, []).
%%   NewProcess is a PID

%%   send message to NewProcess
NewProcess ! {self(), 'hello world'}.
```

`self()` in tuple `{self(), 'hello world'}` identifies the sending process

Message Receipt, I

Syntax for message receipt is much like `case` statement:

```
receive
    msg_pattern1 -> body1 ;
    msg_pattern2 -> body2 ;
        ...
    msg_patternN -> bodyN
end.
```

Example

Original process:

```
NewProc = spawn(msg_example, receiver, []),
NewProc ! {self(), 'hello world'}.
```

Spawned process:

```
-module(msg_example).
-export( [ receiver/0 ] ).

receiver() -> rcv_loop().

rcv_loop() ->
   receive
      {Sender, Msg} ->
         io:fwrite("Received ~p from ~s~n", [Msg, Sender]),
         rcv_loop()
   end.
```

Message Receipt, II

- `receive` watches process

- If some message in mailbox matches one of the receive patterns, the first such message causes the corresponding body to execute

- If no message in mailbox matches one of the receive patterns, `receive` blocks until a message arrives that matches one of the patterns