FUN with FUNctions

Mark Greenstreet

CpSc 418 – January 8, 2016

Outline:

Objectives

- Learn to use pattern matching to express the expected “shape” of data
- Learn how higher-order functions provide abstractions for common control patterns
  - common examples: `map`, `filter`, `fold`
- Learn how list-comprehensions provide a compact way to express `map` and `filter`.
- Learn how recursive functions execute and the implications for writing correct, and efficient code.

A Running Example

- Let's say we have a list `L`. We want to produce a list `C` that gives the number of occurrences of each element of `L`. For example,

```
occurrences([a, b, c, d, b, e, a, q, b, f, e, s]) ->
[{a,2}, {b,3}, {c,1}, {d,1}, {e,2}, {f,1}, {q,1}, {s,1}].
```

  The tuples in the result list can be in any order.
- Strategy:
  1. Sort `L` to produce `S`. Duplicated elements of `L` appear consecutively in `S`.
  2. Traverse `S` counting consecutive occurrences of elements and building `C`.
- Design:
  - The "traverse" step needs a function to recursively walk `S`.
  - I'll call this function `traverse(S, C)`.
  - `traverse` builds `C` as it walks down `S`.

Example: the code

```
occurrences(L) ->
    S = lists:sort(L),
    C = traverse(S, What_goes_here?),
    C.

    traverse(S, C) -> ....
```

- How about an empty list for the "initial" value for `C`?

Example: the code

```
occurrences(L) ->
    S = lists:sort(L),
    C = traverse(S, []),
    C.

    traverse(S, C) -> ....
```

- Now, we need to write `traverse`. I'll initially write it using `if`; so we can compare with a pattern matching approach later.

Erlang's `if`

- Erlang's `if` is a multi-way selection:

```
if cond1 -> expr;
   cond2 -> exp2;
   ...condn -> expn
end
```

- Erlang tries the `cond1` through `condn` expressions in order.
- If one evaluates to `true`, the corresponding expression is evaluated, and that's the value for the `if` expression.
- If no condition is true, then an error is thrown – every expression must have a value.
- For more see slide 32 or "What the If" in *Learn You Some Erlang*.

Writing `traverse` using `if`

```
traverse(S, C) ->
    if % what should the first condition be?




    end.
```

- We plan to recurse along `S`.
- `S == []` seems like a good choice for the first case.
- In this case, `C` has all the occurrence counts. We return it.
- Examples:

```
traverse([], []) -> [];
traverse([], [{a,1}, {b,3}, {c,2}]) ->
    [{a,1}, {b,3}, {c,2}].
```

Writing `traverse` using `if`

```
traverse(S, C) ->
    if S =:= [] -> C;
        % now what?




    end.
```

- The remaining cases have `S` non-empty.
- If `C` is empty,
  - Then, the head of `S` is a new value, we'll prepend it to `C` with a count of 1.
  - Example:

```
traverse([a, b, b, b, c, c], []) ->
    traverse([b, b, b, c, c], [{a,1}]).
```

Writing `traverse` using `if`

```
traverse(S, C) ->
   if S =:= [] -> C;
      C =:= [] -> traverse(tl(S), [{hd(S), 1}]);
      % now what?




   end.
```

- The remaining cases have `S` and `C` non-empty.
- We compare the element at the head of `S` with the "value" part of the tuple at the head of `C`.
  - If they match, we add one to the count for the tuple at the head of `C`.
- Example:

```
traverse([b, b, c, c], [{b,1}, {a,1}]) ->
   traverse([b, c, c], [{b,2}, {a,1}]);
```

Writing `traverse` using `if`

```
traverse(S, C) ->
   if S =:= [] -> C;
      C =:= [] -> traverse(tl(S), [{hd(S), 1}]);
      hd(S) =:= element(1, hd(C)) ->
         traverse(tl(S), [setelement(2, hd(C)
                                    element(2, hd(C))+1) | tl(C)]);
      % now what?

   end.
```

- The remaining case have `S` and `C` non-empty, and the head of `S` doesn't match the value part of the tuple at the head of `C`.
  - We create a new tuple for the head of `C`.
- Example:

```
traverse([c, c], [{b,3}, {a,1}]) ->
   traverse([c], [{c,1}, {b,3}, {a,1}]);
```

Writing `traverse` using `if`

```
traverse(S, C) ->
   if S =:= [] -> C;
      C =:= [] -> traverse(tl(S), [{hd(S), 1}]);
      hd(S) =:= element(1, hd(C)) ->
         traverse(tl(S), [setelement(2, hd(C)
                                  element(2, hd(C))+1) | tl(C)]);
      hd(S) =/= element(1, hd(C)) ->
         traverse(tl(S), [{hd(S), 1} | C])
   end.
```

- That's all.
- Slide slide 11 shows an implementation of `traverse` using pattern matching.

Let's try it!

```
1> c(examples).
{ok,examples}
2> examples:occurrences([a, b, c, d, b, e, a, q, b, f, e, s]).
[{s,1},{q,1},{f,1},{e,2},{d,1},{c,1},{b,3},{a,2}]
```

- Yay – it works!!! (for one test case)
- But the tuples are backwards by their "value" component.
  - That's OK – the problem description on slide 3 said the tuples can be in any order.
  - We'll come back to this later.
- Now, we're ready to see how we could use pattern matching with this problem.

Pattern Matching

- Erlang makes extensive use of pattern matching.
  - The examples on this slide are very simple because of the small Erlang fragment that we have so far.
  - More extensive examples will occur on subsequent slides.
- Simple example:

```
3> [Head | Tail] = [1,5,34].
[1,5,34]
4> Head.
1
5> Tail.
[5,34]
6> [X, Y] = [1,4,34].
** ...:  no match of right hand side value [1,5,34]
```

- `Head` and `Tail` were unbound before executing command 3.
- The Erlang run-time finds if there is a way to choose values for `Head` and `Tail` such that the left side of the `=` operator, `[Head | Tail]`, matches the right side, `L1`.
- The Erlang run-time finds such a choice of values and sets `Head` and `Tail` accordingly.
- If there's no way to make a match, then an error is reported.

More Matching

- The general form for matching is: *LeftSide* = *RightSide*.
- *LeftSide* can be an expression of Erlang values and unbound variables combined using lists and tuples.
- *RightSide* can be an arbitrary expression.
- If a variable name begins with an underscore, the value is ignored.
- Examples:

```
7> [1 | X1] = L1.
[1,5,34]
8> X1.
[5,34]
9> [A1, B1, 2*17] = L1. % The compiler replaces 2*17 with 34.
[1,5,34]
10> [1, 5, 2*C1] = L1.
* 1: illegal pattern % But it's not a general equation solver!
11> [_, B2, _] = L1.
[1,5,34]
12> B2.
5
```

- See also: Pattern Matching in *LYSE*.

Writing `traverse` using pattern matching

```
traverse([], C) -> C;
traverse([H | T], []) -> traverse(T, [{H, 1}]);
% the rest is coming soon.
```

- The first clause handles the case when `S` is empty.
- The second clause handles the case when `C` is empty.

Writing `traverse` using pattern matching

```
traverse([], C) -> C;
traverse([H | T], []) -> traverse(T, [{H, 1}]);
traverse([H | T], [{H, N} | C_T]) ->
   traverse(T, [{H, N+1} | C_T]);
% one more clause to write.
```

- The first clause handles the case when `S` is empty.
- The second clause handles the case when `C` is empty.
- We now need to handle the case when both `S` and `C` are non-empty.
  - This means comparing the head of `S` with the value part of the tuple at the head of `C`.
  - If they match, we'll increment the count of `C`'s tuple.
  - Otherwise, we'll prepend a new tuple to `C`.

Writing `traverse` using pattern matching

```
traverse([], C) -> C;
traverse([H | T], []) -> traverse(T, [{H, 1}]);
traverse([H | T], [{H, N} | C_T]) ->
   traverse(T, [{H, N+1} | C_T]);
traverse([H | T], C) ->
   traverse(T, [{H, 1} | C]).
```

- The first clause handles the case when `S` is empty.
- The second clause handles the case when `C` is empty.
- The third clause handles the case when the head of `S` matches the value part of the tuple at the head of `C`.
  - By using `H` in the pattern for `S` **and** in the pattern for `C`, we make sure that they match.
- The fourth clause handles the case when they head of `S` doesn't correspond to the head of `C`.
  - In this case, we just prepend a new tuple to `C`.

Compare the two versions

```
traverse(S, C) ->
   if S =:= [] -> C;
      C =:= [] -> traverse(tl(S), [{hd(S), 1}]);
      hd(S) =:= element(1, hd(C)) ->
        traverse(tl(S), [setelement(2, hd(C)
                                    element(2, hd(C))+1) | tl(C)]);
      hd(S) =/= element(1, hd(C)) ->
        traverse(tl(S), [{hd(S), 1} | C])
end.
```

```
traverse([], C) -> C;
traverse([H | T], []) -> traverse(T, [{H, 1}]);
traverse([H | T], [{H, N} | C_T]) ->
   traverse(T, [{H, N+1} | C_T]);
traverse([H | T], C) ->
   traverse(T, [{H, 1} | C]).
```

- Use pattern matching instead of `if` when patterns are clearer.
- That's most of the time.

Higher-Order Functions

- In Erlang, functions are “first-class citizens”:
  - Functions can be bound to variables.
  - Functions can be passed as parameters to other functions.
  - A function can return a function as its result.
- Java and other languages provide similar mechanisms.
  - For example, sorting methods that can take a “comparator” object as a parameter.
  - Other examples include call backs, and event handlers
- In functional programming, higher-order functions are used:
  - To express common programming patterns.
  - and all the reasons mentioned above.

`fun` Expressions

Erlang provides two mechanisms for creating expressions with function values:

- `fun` *Module*`:`*Function*`/`*Arity* where
  - `Module` is an atom, the name of a module.
  - `Function` is an atom the name of a function exported by *Module*.
  - `Arity` is the number of arguments of function *Module*`:`*Function*.
- `fun (`*Args*`) -> `*Expr*` end`
  - Example:

```
13> Plus = fun(X, Y) -> X+Y end.
#Fun<erl_eval.12.54118792>
14> Plus(2, 3).
5
```

- See also Anonymous Functions in *LYSE*.

Some higher-order functions

From the `lists` module in the standard Erlang library.

- `lists:map(Fun, List1) -> List2`
  - Apply `Fun` to each element of `List1` to produce `List2`.
  - Example:

```
15> L = [1, 2, 3].
[1,2,3]
16> lists:map(fun(X) -> 2*X end, L).
[2,4,6]
```

- `lists:foldl(Fun, Acc0, List) ->Acc1`
  - Combine the elements of `List` using `Fun` – work from the left to right.
  - `Fun` takes two arguments:
    - The first is the current element from `List`.
    - The second is the current value of the accumulator.
  - Example:

```
17> lists:foldl(Plus, 0, L).
% Equivalent to: Plus(3, Plus(2, Plus(1, 0))).
6
```

- See also Maps, filters, folds, and more in *LYSE*.

Another example of `foldl`

Converting strings to decimal:

```
str_to_int(S) ->
   lists:foldl(fun (C, N) -> 10*N + (C - $0) end, 0, S).
```

Note that `$0` is the character that prints as '0'.

- I put this in the `examples` module. Let's try it.

```
18> c(examples).
{ok,examples}
19> examples:str_to_int("123").
123
```

Implementing `traverse` with `foldl`

```
traverse(S) ->
    lists:foldl(fun examples:traverse_help/2, ??, S).
traverse_help(A, ?)  ->
    ???
```

- What should the “accumulator” be?

Implementing `traverse` with `foldl`

```
traverse(S) ->
    lists:foldl(fun examples:traverse_help/2, [], S).

traverse_help(A, C) ->
    ???
```

- What should the "accumulator" be?
- Let's make the accumulator be the `C` list from our earlier version.
  - The initial value of the accumulator is `[]`.

Implementing `traverse` with `foldl`

```
traverse(S) ->
    lists:foldl(fun examples:traverse_help/2, [], S).

traverse_help(A, C) ->
    ???
```

- Let's make the accumulator be the `C` list from our earlier version.
  - The initial value of the accumulator is `[]`.
- What should the body of `traverse_help` be?

Implementing `traverse` with `foldl`

```
traverse(S) ->
    lists:foldl(fun examples:traverse_help/2, [], S).

traverse_help(A, []) -> [{A,1}];
traverse_help(A, [{A,N} | T]) -> [{A,N+1} | T];
traverse_help(A, C) -> [{A,1} | C];
```

- What should the body of `traverse_help` be?
- We can just consider the same cases as we had for the pattern matching version of traverse:
  - If `C` is empty, make a singleton list.
  - If `A` matches the value-part of the head of `C`, increment the count.
  - Otherwise, prepend a new tuple to `C`.

Some more higher order functions

- `lists:filter(Pred, List1) -> List2`
  - Return a list of the elements of `List1` that satisfy `Pred`.
  - Example: `divisible_drop` (from `examples.erl` from the Jan. 6 lecture).

```
divisible_drop(N, [A, Tail]) ->
   if A rem N == 0 -> divisible_drop(N, Tail);
   A rem N /= 0 -> [A | divisible_drop(N, Tail)]
end.
```

  - `divisible_drop` using `lists:filter`

```
divisible_drop(N, L) ->
   lists:filter(fun (A) -> A rem N =/= 0 end, L).
```

- We'll see many more examples of higher order functions as we continue to use Erlang.

List Comprehensions

- Map and filter are such common operations, that Erlang has a simple syntax for such operations.
- It's called a **List Comprehension**:
  - `[`*Expr* `||` *Var* `<-` *List*`,` *Cond*`, ...]`.
  - *Expr* is evaluated with *Var* set to each element of *List* that satisfies *Cond*.
  - Example:

```
20>R = count3s:rlist(5, 1000).
[444,724,946,502,312].
21>[X*X || X <- R, X rem 3 == 0].
[197136,97344].
```

- See also List Comprehensions in *LYSE*.

List Comprehensions – More Examples

- Doubling the elements of a list (compare with slide 15 where `L` is bound to `[1,2,3]`).

```
22> [2*X || X <- L]. [2,4,6].
```

- Yet another version of `divisible_drop`

```
divisible_drop(N, L) ->
   [A || A <- L, A rem N =/= 0].
```

Head vs. Tail Recursion

- I wrote two versions of computing the sum of the first `N` natural numbers:

```
sum_h(0) -> 0; % "head recursive"
sum_h(N) -> N + sum_h(N-1).

sum_t(N) -> sum_t(N, 0).
sum_t(0, Acc) -> Acc; % "tail recursive"
sum_t(N, Acc) -> sum_t(N-1, N+Acc).
```

- Here are some run times that I measured:

N	t_{head}	t_{tail}	N	t_{head}	t_{tail}
1K	21μs	13μs	1M	21ms	11ms
10K	178μs	114μs	10M	1.7s	115ms
100K	1.7ms	1.1ms	100M	28s	1.16s
			1G	$>$ 8 min	11.6s

Head vs. Tail Recursion – Comparison

- Both grow linearly for $\mathtt{N} \leq 10^6$.
  - The tail recursive version has runtimes about 2/3 of the head-recursive version.
- For $\mathtt{N} > 10^6$,
  - The tail recursive version continues to have run-time linear in $\mathtt{N}$.
  - The head recursive version becomes much slower than the tail recursive version.
  - Additional note: the head-recursive timings were very hard to reproduce – I'll explain that shortly.

Tail Call Elimination

- The Erlang compiler optimizes the case that the last operation in a function is another function call (recursive or otherwise).
  - If this were handled in the traditional way, the call would create a new stack frame, and when the called function returned, this function would copy the return value to the next frame, and return.
  - Instead, the compiler creates code to **reuse** the current stack frame for the newly called function, but the return address remains the same. When the newly called function returns, the return will "skip over" the functions that ended with tail calls.
  - The compiler has turned the recursive function into a while-loop.
  - Conclusion: **When people tell you that recursion is slower than iteration – don't believe them.**
- The head recursive version creates a new stack frame for each recursive call.
  - I was hoping to run my laptop out of memory and crash the Erlang runtime – makes a fun, in-class demo.
  - But, OSX does memory compression. All of those repeated stack frames are very compressible. This creates the crazy runtime. Compression slows the process down dramatically, but it goes faster once the OS has "learned" the patterns.

Tail Call Elimination – a few more notes

- Can you count on your compiler doing tail call elimination:
  - In Erlang, the compiler is **required** to perform tail-call elimination. We'll see why on Monday.
  - In Java, the compiler is forbidden from performing tail-call elimination. This is because the Java security model involves looking back up the call stack.
  - `gcc` performs tail-call elimination when the `-o` flag is used.
- Is it OK to write head recursive functions?
  - Yes! Often, the head-recursive version is much simpler and easier to read. If you are confident that it won't have to recurse for millions of calls, then write the clearer code.
  - Yes! Not all recursive functions can be converted to tail-recursion.
    - ★ Example: tree traversal.
    - ★ Computations that can be written as "loops" in other languages have tail-recursive equivalents.
    - ★ But, recursion is more expressive than iteration.

Summary (1/3)

- Pattern Matching
  - An expressive way to write functions according to the shape of the data they process.
  - Patterns are usually much clearer than lots of `if`s and functions like `hd`, `tl`, and `element` to walk the data structure.
  - Use patterns when they produce clearer coded.
    - ★ Grading will include marks for good coding including: clarity, test cases, helpful comments.
- Higher-order functions
  - Higher-order functions provide abstractions for common programming patterns such as map, fold, and filter.
  - Anonymous functions, `fun(`*Args* `->` *Expr* `end`, allow you to write create function-valued terms (esp. arguments to map, etc.), where they are used.

Summary (2/3)

List Comprehensions

- Common map and filter operations can be written succinctly as list-comprehensions.
- Unless specifically stated, I won't require you to write them – this isn't a functional programming course (that's CpSc 312).
- I will use list comprehensions in code I use as examples – you'll need to be able to read them.

Summary (3/3)

Tail Call elimination

- Tail call elimination avoids creating a new stack frame when the last operation of a function is to return the value produced by another function call.
- Tail recursive functions are usually a bit faster and use less memory than their head-recursive counterparts.
- This is especially important if the recursion could be very deep (or unbounded).
- On the other hand, head-recursive functions are often simpler to write.
  - Writing simple and clear code is good.
  - Unless there is a good reason to believe that the optimizations are needed.
  - “premature optimization is the root of all evil” – D.E. Knuth in Computer Programming as an Art.

Preview

January 11: Processes and Messages

Reading: *Learn You Some Erlang*, The Hitchhiker's Guide ... through More on Multprocessing

Mini-Assignment: Mini-Assignment 2 due **10:00am**

January 13: Reduce and Scan (simple)

Reading: Lin & Snyder, chapter 5, pp. 112–125

January 15: Reduce and Scan (generalize)

Homework: Homework 1 deadline for early-bird bonus (11:59pm)

Homework 2 goes out – parallel programming with Erlang

January 18: Architecture Review

Reading: Pacheco, Chapter 2, Sections 2.1 and 2.2.

Homework: Homework 1 due **11:59pm**

January 20: Shared-Memory Machines

Reading: Pacheco, Chapter 2, Section 2.3

January 22: Distributed-Memory Machines

Reading: Pacheco, Chapter 2, Sections 2.4 and 2.5.

Review Questions (1/2)

- The function, `uniq`, below replaces all (non-empty) subsequences of the same value in a list with just one occurrence of that value. For example,
  `uniq([a,b,b,c,a,a,a,d]) -> [a,b,c,a,d]` Here's the code written with `if`:

```
uniq(L) ->
   if length(L) =< 1 -> L;
      hd(L) == hd(tl(L)) -> uniq(tl(L));
      true -> [hd(L) | uniq(tl(L))]
end.
```

  Write a new version of `uniq` using pattern matching.
- Consider the `count3s` function from the Jan. 4 lecture:

```
count3s([]) -> 0;
count3s([3 | Tail]) -> 1 + count3s(Tail);
count3s([_Other | Tail]) -> count3s(Tail).
```

  - Write a new version of `count3s` using `lists:filter` and `length`.
  - Write a new version of `count3s` using a list comprehension and `length`.

Review Questions (2/2)

- What is a higher-order function?
- Write a function
  `test_it(F1, F2, TestCases)` where `F1` and `F2` are functions with one parameter, and `TestCases` is a list. The `test_it` function evaluates `F1` and `F2` on each element of `TestCases` and returns `true` if they agree on all cases and `false` otherwise.
  - Is `test_it` a higher order function? Why or why not?
  - How can you use list comprehensions and/or higher-order functions such as `and`, `filter`, and `foldl` in your implementation?
- What is the tail recursion?
  - Write `count3s` using tail recursion.
  - Describe a situation where it is very important to use tail recursion.
  - Describe a situation where it is better not to use tail recursion.

Supplementary Material

The remaining slides are some handy material that we won't cover in lecture, but you can refer to if you find it helpful.

- `if`
- `when`
- `case`
- A guide to Erlang Punctuation
- Table of Contents.

if

- As noted on slide 5, an Erlang `if` expression throws an error if none of the conditions are satisfied.

```
1> X = 0.
2
2> if X < 0 -> hello;
2>    X > 0 -> world
2> end.
** exception error:  no true branch found ...
```

  - We can add a "catch-all" by including `true` as the condition for the default case:

```
3> if X < 0 -> hello;
3>    X > 0 -> world;
3>    true  -> 'eh?'
3> end.
'eh?'
```

  - Many Erlang programmers see "`true` means else" as tacky.
    - ★ If there is a condition that is relevant to understanding the final case, say it. For example, `X == 0 -> 'eh?'`

- Guards often take the role of `if` – see the next slide.

When clauses – example

Consider: `examples:sum_t(1.5).`

- It runs forever:

```
sum_t(1.5)-> sum_t(1.5, 0) -> sum_t(0.5, 1.5)
          -> sum_t(-0.5, 2.0) -> sum_t(-1.5, 1.5) -> ...
```

- We want `sum_t` to throw an error if it is called with an argument that isn't a non-negative integer.

```
sum_g(N) when is_integer(N) and (N >= 3) -> sum_t(N, 0).
```

  The '`g`' is for the version with a guard.

- Let's try it:

```
1> c(examples).
{ok,examples}
2> examples:sum_g(3).
6
3> examples:sum_g(1.5).
** exception error:  no function clause matching
       examples:sum_g(1.5) (examples.erl, line 87)
```

When clauses

- Syntax: `when` *Guard*
- *Guard* is a boolean-valued expression
  - The guard can consist of constants, variables, arithmetic and boolean operations, and comparisons.
  - Erlang is restrictive about what functions you can use.
    - ★ built-in functions that have no side-effects.
    - ★ some handy ones: `length(List)`, `element(N, Tuple)`, `is_integer(X)`, `is_list(X)`, `is_tuple(X)`, ...
- See also Guards, Guards! in *LYSE*.

case

A `case` expression just does pattern matching.

- Example:

```
4> X = lists:seq(2,5).
[2,3,4,5]
5> case X of
5>   [] -> io:format("X is an empty list n");
5>   _ when is_list(X)-> io:format("X has  p elements n", [
5>   _ -> io:format("X is not a list")
5> end.
X has 4 elements
ok
```

The `ok` is the return value from `io:format`.

`if` is subsumed by `case`

```
if Cond1 -> Expr1;
   ...
   Condn -> Exprn
end.
```

is equivalent to

```
case ok of
   _ when Cond1 -> Expr1;
   _ when ...;
   _ when Condn -> Exprn
end.
```

- Use `if` when
  - the selection criteria are based on the `value` of the data
  - it doesn't make sense to include the criteria as guards of the function clauses.
- Use `case` when
  - the selection criteria are primarily based on the `shape` of the data
  - include a guard (i.e. `when`) if the value matters as well.

`case` is subsumed by `fun`

```
case Expr of
    Pattern1 -> Expr1;
    ...
    PatternN -> ExprN
end.
```

is equivalent to

```
(fun (Pattern1) -> Expr1;
 (...)  -> ...;
 (PatternN) -> ExprN
 end
)(Expr)
```

- But the `case` form is almost always clearer.
- OTOH, the Erlang compiler implements `if` and `case` by converting them to `fun`.
  - This is where many of the compiler-generate function names come from that you'll sometimes see in a stack backtrace when an error occurs.

Punctuation

- Erlang has lots of punctuation: commas, semicolons, periods, and `end`.
- It's easy to get syntax errors or non-working code by using the wrong punctuation somewhere.
- Rules of Erlang punctuation:
  - Erlang declarations end with a period: `.`
  - A declaration can consist of several alternatives.
    - ★ Alternatives are separated by a semicolon: `;`
    - ★ Note that many Erlang constructions such as `case`, `fun`, `if`, and `receive` can have multiple alternatives as well.
  - A declaration or alternative can be a block expression
    - ★ Expressions in a block are separated by a comma: `,`
    - ★ The value of a block expression is the last expression of the block.
  - Expressions that can consist of multiple alternatives end with `end`
    - ★ `case` *Alternatives* `end`
    - ★ `fun` *Alternatives* `end`
    - ★ `if` *Alternatives* `end`
    - ★ `receive` *Alternatives* `end`

Table of Contents