Reading 7

# Exercise 1: Summarize

Meta Language (ML) uses patterns to define functions and manage data structures, while let and val expressions handle variable scope, with practical applications such as merge sort that can handle multiple inputs or varying function requirements.

# Exercise 2: Read & Code

fun f \_ = "yes"; (\* Function f that ignores its input and returns "yes" \*)

f 34.5; (\* Calls f with 34.5, returns "yes" \*)

f []; (\* Calls f with an empty list, returns "yes" \*)

fun f 0 = "yes"; (\* Function f that matches 0 and returns "yes" \*)

f 0; (\* Calls f with 0, returns "yes" \*)

f 0; (\* Calls f with 0 again, returns "yes" \*)

f 1; (\* Calls f with 1, raises a match failure exception because f is only defined for 0 \*)

fun f [a,\_] = a; (\* Function f that matches a list with at least two elements and returns the first element \*)

f [#"f",#"g"]; (\* Calls f with a list of two characters, returns #"f" \*)

fun f (x::xs) = x; (\* Function f that matches a non-empty list and returns the first element \*)

f [1,2,3]; (\* Calls f with a list of integers, returns 1 \*)

fun f 0 = "zero" (\* Function f that matches 0 and returns "zero" \*)

| f 1 = "one"; (\* Matches 1 and returns "one" \*)

f 1; (\* Calls f with 1, returns "one" \*)

fun f 0 = "zero" (\* Function f that matches 0 and returns "zero" \*)

| f \_ = "non-zero"; (\* Matches any other integer and returns "non-zero" \*)

f 0; (\* Calls f with 0, returns "zero" \*)

f 34; (\* Calls f with 34, returns "non-zero" \*)

fun eq (a,b) = if a=b then 1 else 0; (\* Function eq that checks if two values are equal, returns 1 if true, 0 if false \*)

eq (1,3); (\* Calls eq with 1 and 3, returns 0 \*)

eq ("abc","abc"); (\* Calls eq with two identical strings, returns 1 \*)

fun halve nil = (nil, nil) (\* Function halve that splits an empty list into two empty lists \*)

| halve [a] = ([a], nil) (\* Splits a single-element list into a list with that element and an empty list \*)

| halve (a::b::cs) = (\* Splits a list with at least two elements \*)

let

val (x, y) = halve cs (\* Recursively halve the rest of the list \*)

in

(a::x, b::y) (\* Combine the results \*)

end;

halve [1]; (\* Calls halve with a single-element list, returns ([1],[]) \*)

halve [1,2]; (\* Calls halve with a two-element list, returns ([1],[2]) \*)

halve [1,2,3,4,5,6]; (\* Calls halve with a six-element list, returns ([1,3,5],[2,4,6]) \*)

fun merge (nil, ys) = ys (\* Function merge that merges two lists, returns the second list if the first is empty \*)

| merge (xs, nil) = xs (\* Returns the first list if the second is empty \*)

| merge (x::xs, y::ys) = (\* Merges two non-empty lists \*)

if (x < y) then x :: merge(xs, y::ys) (\* Adds the smaller element to the result and continues merging \*)

else y :: merge(x::xs, ys);

merge ([2],[1,3]); (\* Calls merge with two lists, returns [1,2,3] \*)

merge ([1,3,4,7,8],[2,3,5,6,10]); (\* Calls merge with two lists, returns [1,2,3,3,4,5,6,7,8,10] \*)

fun mergeSort nil = nil (\* Function mergeSort that sorts an empty list \*)

| mergeSort [a] = [a] (\* Returns a single-element list as is \*)

| mergeSort theList = (\* Sorts a list with more than one element \*)

let

val (x, y) = halve theList (\* Split the list into two halves \*)

in

merge(mergeSort x, mergeSort y) (\* Recursively sort and merge the halves \*)

end;

mergeSort [4,3,2,1]; (\* Calls mergeSort with a list, returns [1,2,3,4] \*)

mergeSort [4,2,3,1,5,3,6]; (\* Calls mergeSort with a list, returns [1,2,3,3,4,5,6] \*)

# Exercise 3: Inquire

Elixir and ML share similarities in their pattern matching by being able to define multiple clauses in a single function. They are executed in order and can include a catch-all pattern (“\_”) to handle any remaining or unforeseen inputs.

Example:

ML:

fun road "high" = "You take the high road!"

| road "low" = "I'll take the low road! (and I'll get there before you)"

| road \_ = "Take the 'high' road or the 'low' road, thanks!"

Elixir:

iex> road = fn

"high" -> "You take the high road!"

"low" -> "I'll take the low road! (and I'll get there before you)"

\_ -> "Take the 'high' road or the 'low' road, thanks!"

end

Both pattern matching functions are similar in structure, with ML being more concise in its implementation. Pattern matching in both languages allows for clear function definitions that can handle multiple input patterns.