Programming languages: Erlang patterns

Introduction

In these notes, more advanced features of Erlang will be introduced. Specifically, the notion of specifying *patterns* for inputs will simplify the specification of functions. The use of local variables and multi-statement expressions will also be discussed within the context of improving the efficiency and readability of functions.

Review

Previously, a function to *reverse a list* was used as an example of:

- defining functions
- making use of the *if-else* syntax
- making use of the syntax to deal with the list data structure
- making use of recursion

That function was specified as:

```
reverse(L) ->
if
    L == [] ->
            [];
    true ->
            reverse(tl(L)) ++ [hd(L)]
end.
```

and a sample of execution was:

```
erl1:reverse([1,2,3]).
[3,2,1]
```

Patterns

Erlang provides an alternative formulation for defining certain types of if-else driven functions which tends to be more intuitive than the default *if-else* formulation. This alternative makes use of patterns in the function argument list. This is a feature NOT available in other languages you are aware of (such as C++ and Java).

A pattern in Erlang is an expression over a set of variables. This pattern is placed in the argument list when a function is defined. When an actual parameter value matches the pattern at runtime, the variables listed inside the pattern are assigned the appropriate actual parameter values. These variables can then be used directly within the function.

As an example, one commonly used pattern is the expression *[X|XS]*. This pattern indicates an element named *X* being attached to a list named *XS*. Accordingly this pattern will match any list with one or more elements (anything but the empty list). As an example, the list [1] matches this pattern as [1| [ ]], and the list [1,2,3] matches as [1 | [2,3]]. A programmer can use more than two variables in a pattern.

To define a function using patterns, a function definition is attached to each possible set of arguments for that function. Semicolons are used to separate the multiple patterns allowed for the function and to delineate the different possible code-paths through the function to be executed. The last clause ends with a period. The last clause is often designed to use the pattern Other, representing a variable which will match any input, and is implemented to handle a default case. The general syntax for a pattern based function definitions is as follows:

```
functionName(<first pattern>) ->
    <first result expression>;
functionName(<second pattern>) ->
    <second result expression>;
functionName(<third pattern>) ->
    <third result expression>;
...
functionName(<last pattern>) ->
    <last result expression>.
```

The function name has to be the same on every line, but everything else can change (different patterns, different function code).

Erlang will parse through a list of patterns for a function, looking for a match and only executing the first pattern match that is found. Erlang will report an error at runtime (*no function clause matching...*) if an in-exhaustive set of patterns is used and an actual argument sent to a function is unable to be matched.

Example uses of patterns

The following functions provide varying examples of ways that patterns can be used in defining functions.

Reversing a list

With the idea of patterns, the list *reverse* function defined earlier (on page 1) can now be rewritten using patterns:

```
reverse([]) ->
    [];
reverse([X|XS]) ->
    reverse(XS) ++ [X].
```

That is arguably easier to read and more intuitive compared to the original:

```
reverse(L) ->
if
    L == [] ->
            [];
    true ->
            reverse(tl(L)) ++ [hd(L)]
end.
```

Note that the original *if-else*, as well as variable assignment (*hd(L)* to *X*, *tl(L)* to *XS*), are performed implicitly with this pattern approach.

Factorial

Defining factorial provides another example of the use of patterns, this time with the patterns representing integer values.

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

Merge two sorted lists

Here's another example of pattern matching - a *merge* function which merges two integer lists in sorted order, given the lists themselves are already in sorted order. This would be used as part of mergesort – many of you have likely heard of mergesort in CSC 112 or CSC 221.

```
merge(List1,[]) ->
    List1;
merge([],List2) ->
    List2;
merge([X|XS],[Y|YS]) ->
    if (X < Y) ->
        [X | merge(XS,[Y|YS])];
    true ->
        [Y | merge([X|XS],YS)]
    end.
```

The base cases for this function are for handling a merge of a non-empty list with an empty list. The recursive case determines which head entry is smaller. That smaller entry is attached to the front of a list that is recursively generated by merging the rest of the list that contained the smaller element with the complete other list.

Types of patterns

A list of the most commonly used patterns that can be used as function arguments are:

- Constants: [ ] (for the empty list), 0 (or any raw numerical value)
- List expressions using the operator | : [X|XS], [X | [Y|YS]]

Wildcard pattern

The underscore symbol can be used as a *wildcard* or *don't care* pattern – it matches anything, but since it is not a variable, the value of what was matched against the wildcard cannot be accessed. Here is an example of a list *contains* function that employs pattern matching and the wildcard symbol.

```
contains(_,[]) -> % always false if list empty, regardless of query item
    false;
contains(X,[X|_]) -> % true if query item matches head, regardless of rest of list
    true;
contains(X,[_|YS]) -> % can assume didn't match head (if you got to this line),
% so test against rest of list
    contains(X,YS).
```

If you see a *Warning* that a variable is being unused, it is likely that Erlang is suggesting that it could be changed to a wildcard variable.

Explicit local variables and multiple expressions

The conceptual parts of programming are often easier when one allows the definition of local variables within functions. In the previous notes, variables were only seen as the inputs to functions. Local variables within a function are important in improving the readability (and thus write-ability) of software. Local variables may also increase program performance as operations that must be repeated multiple times (when local variables aren't allowed) to get one piece of data can be collapsed into one call.

Local variables can be used inside of a function definition by using the assignment (=) operation. Importantly, note that *variables can only be assigned a value once within a function call* (but can be re-assigned across calls).

To make effective use of variable assignments, we also need to take advantage of the ability to have more than one expression in a function. A function can be defined as executing more than one expression by separating each expression by a comma. The value of the *last expression* that is executed is the value returned from the function.

Example uses of local variables and multiple expressions

Computing x^{20}

An example of variable assignment and comma separated expressions is shown below in a function for finding x^{20}:

```
power20(X) ->
    Two = X*X, Four = Two*Two, Four*Four*Four*Four*Four.
```

Note that the function first computes x^2 and stores that result in a variable named *Two*, then uses that *Two* variable to compute x^4 and stores that in a different variable named *Four*, and then uses that final variable named *Four* to computer x^{20}.

Splitting a list into two lists

Here's a useful example for variables which performs splitting of a list into two lists – one list contains the elements in the odd positions of the original list, the other contains the elements in the even positions. The returned value is a 2-element *tuple*. A tuple is just an ordered collection of items (a 2-element tuple is a pair). In this example, the first element of the pair is one of the lists that is created (the list of values in odd positions of the original list), the second element of the pair is the other list that is created (the list of values in even positions of the original list).

```
split([]) ->    % an empty list, when split, gives 2 empty lists
    {[],[]};
split([X]) ->  % a 1-element list gives the 1-element list, and an empty list
    {[X],[]};
split([X|[Y|YS]]) -> % handle multi-element by breaking odd/even elements
    {M,N} = split(YS), {[X|M],[Y|N]}.
```

A summary example

We can write a *mergesort* sorting function (one of the faster sorts) in Erlang by combining our previously described split and merge functions:

```
% taken from earlier in the lab
split([]) ->
    {[],[]};
split([X]) ->
    {[X],[]};
split([X|[Y|YS]]) ->
    {M,N} = split(YS), {[X|M],[Y|N]}.

% taken from earlier in the lab
merge(List1,[]) ->
    List1;
merge([],List2) ->
    List2;
merge([X|XS],[Y|YS]) ->
    if (X < Y) ->
        [X | merge(XS,[Y|YS])];
    true ->
        [Y | merge([X|XS],YS)]
    end.

% new code
mergeSort([]) ->
    [];
mergeSort([X]) ->
    [X];
mergeSort(Y) ->
    {M,N} = split(Y), A = mergeSort(M), B = mergeSort(N), merge(A,B).
```