Comparing expressions

Erlang offers two different mechanisms to compare expressions: **matching** and numeric comparisons. The former is found in functional and logic programming, but not in mainstream programming languages. The latter is ubiquitous.

Numeric comparisons involve a comparison operator and two operands. For instance, consider the following equality in both Java and Erlang:

```
fact(5) == 119 + 1
```

Note that

1. The two sides are expressions
2. which are separately evaluated,
3. then the values are compared and the result is either true or false.

Java variable definitions

Consider a Java variable definition like

```
int N = 3;
```

This actually means three different things:

1. define the variable `N`
2. whose type is `int`
3. and initial value is `3`.

Initialisation is a separate concept since the above is equivalent to

```
int N;
N = 3;
```

Matching in Erlang

In Erlang, the programmer does not specify the types, as the compiler does not check most of them, so the above is simply written in Erlang

```
N = 3
```

Erlang provides an extension of the Java-like variable definition: *matching*.

1. The left-hand side must be a **term** (not an expression),
2. the right-hand side is an expression
3. which is evaluated before the matching;
4. the matching is itself an expression whose value (in case of success) is the value of the right-hand side (as in C++ "`a = (b = 1);`").

Matching (cont)

What if `N` does not occur for the first time in the matching? Consider

```
-module(match).
-export([f/1]).

f(N) -> N = 3.
```

In this case, `N` in the matching is replaced first by its value, which is the argument of the function `f` when it is called.

So `match:f(3)` succeeds but, for example, `match:f(2)` fails.

Matching (cont)

Therefore, the rule for evaluating the matching

$$t = e$$

is actually better expressed as follows:

1. replace all occurrences of variables in the term t which are already bound by their value: the new term is t';
2. evaluate expression e into v;
3. match v against t';
4. in case of success, the value of the matching is v.

The left-hand side of a matching (here noted t) is called a **pattern**.

Matching (cont)

Assume that `N` appears for the first time in the matching

```
N = 3
```

Then, it is a simple case of matching since

1. the left-hand side, `N`, being a variable, is a term;
2. the right-hand side, `3`, is an expression (actually a value);
3. the evaluation of the right-hand side is itself, `3`, since it is a value already;
4. the matching of numbers is the same as the numeric equality;
5. the value of the matching is `3`.

Importantly, the matching leads to bind variable `N` to `3`, noted $\texttt{N} \mapsto \texttt{3}$.

Matching (cont)

What is the difference, in Erlang, between `N = 3` and `N == 3`?

The first case is a matching and the second is a numeric comparison.

In the latter, `N` must already be defined (bound) so it can be evaluated before the comparison takes place. The result if either the atom `true` or `false`.

In the former, if `N` is already bound, the matching either fails or succeeds and its value is `3`.

If `N` is not already bound, then the matching succeeds with the value `3` and the binding of `N` to 3, noted `N` $\mapsto$ 3.

Matching (cont)

You may wonder now why bother? What is the need of matching when we have equality and other comparisons?

Actually the fist difference is the possible **failure**. Consider

```
-module(match).
-export([f/1]).

f(N) -> N = 3.
```

```
-module(eq).
-export([f/1]).

f(N) -> N == 3.
```

The calls to `match:f` either fail, therefore stop the whole program, or return the integer 3. The calls to `eq:f` return either `true` or `false`.

Matching (cont)

The main difference between matching and equality appears when matching complex terms, involving lists and tuples. We will explain this from page 55 on.

Remember also that if `N` is not bound, then

```
N = 3
```

acts as a variable definition.

This way, we can think of the matching as a way to combine both definition and comparison.

Matching/Basic examples

Here is another example:

```
N = 3 + 1
```

Here, the right-hand side is not a term, it is evaluated into `4` before the matching takes place.

Another one:

```
N = 2 * f(3)
```

Here, the function `f` must be in the scope, i.e. must be bound at this point.

Scope and multiple clauses

How are variables defined (i.e. bound) by matching used latter? In other words: what is the scope? In C++, the scope is the definition point until the end of the current bloc. For example

```
...
{
  ...
  int n = 3;
  ...  // ‘n’ is bound here.
}
... // ‘n’ is unbound here.
```

In Erlang, function bodies can be made of series of expressions separated by commas. For example `f(X) -> X=3, X+1.`

Scope and multiple clauses (cont)

The expressions are evaluated from left to right and, as exemplified in

```
f(X) -> X=3, X+1.
```

the bindings created by matches (a match is a successful matching) are accessible for the following expressions. This is why `X` is bound to the value 3 in the expression `X + 1`.

Note that only the value of the last expression becomes the return value of the function. Thus, for example, the value of `X=3`, which is 3, is discarded, **but not the binding of `X`!** Remember that evaluating a match yields to

1. a value
2. *and*, if variables are present on the left-hand side, bindings for these variables.

Matching/Constants

Two numbers match if they represent the same mathematical number, e.g.

```
37 = 37
```

Two atoms match if they are made of the same characters, e.g.

```
abc = abc
'hello world' = 'hello world'
```

A variable matches any term that can be evaluated, e.g.

```
X = 37
X = Y
```

Matching/Tuples

A tuple t matches an expression e, that is

$$t = e$$

if and only if

1. the replacement of the variables of t that are bound yields the (term) tuple t';
2. the evaluation of e yields the tuple v;
3. both t' and v have the same arity;
4. the n-th component of t matches the n-th component of v.

In case of success, the resulting bindings of all sub-matches are joined.

Matching/Tuples (cont)

The matching of tuples is very useful, because it allows to destructure the return values of functions in a single expression.

For example

```
go(Job) ->
  {Status,NewJob,Priority} = launch(Job),
  forward(Message),
  schedule(NewJob,Priority).
```

The tuples can be arbitrarily nested:

```
go(Job) ->
  {{status,{Answer,Log}},Version} =
    {run(Job),version(Job)}.
```

Matching/Lists

A list l matches an expression e, that is

$$l = e$$

if and only if

1. the replacement of the variables in l which are bound yields the (term) list l';
2. the evaluation of e yields the list v;
3. both e and v are the empty list [] or
4. the head of l' matches the head of v and
5. the tail of l' matches the tail of v.

Matching/Lists/Examples

Consider for instance

```
[Head | Tail] = [1,a,2,b,c]
```

This match yields the value `[1,a,2,b,c]` because of the matches

```
Head = 1
Tail = [a,2,b,c]
```

The bindings are thus `Head` $\mapsto$ `1` and `Tail` $\mapsto$ `[a,2,b,c]`.
Consider also the two matchings

```
{X,Y} = {a,b}, [A,X|B] = [Y,a]
```

which succeed, yielding the bindings `X` $\mapsto$ `a`, `Y` $\mapsto$ `b`, `A` $\mapsto$ `b` and `B` $\mapsto$ `[]`.

Matching/Mixed examples

The matching

```
{C, [Head | Tail]} = {{222, man}, [a,b,c]}
```

succeeds with the bindings {`C` ↦ {`222, man`}, `Head` ↦ `a`, `Tail` ↦ `[b,c]`}.

The matching

```
[{person,Name,Age,_}|T] = [{person,fred,22,male}]
```

succeeds with bindings {`Name` ↦ `fred`, `Age` ↦ `22`, `T` ↦ `[]`}. Note the pattern "_" which stands for an unknown but unique variable.

Matching/Non-linear patterns

Remember that a matching consists in comparing structurally the value of an expression to a term, called a pattern.

It is possible in Erlang to repeat a variable in a pattern, in order to save another match later. For example

```
{X,X} = {a,f()}
```

is equivalent to

```
{X,Y} = {a,f()}, X = Y
```

The matching

```
{A, foo, A} = {123, foo, abc}
```

fails.

Matching/Calling functions

The Erlang functions are called using pattern matching too. At page 18 we defined the factorial function as

```
-module(math1).
-export([fact/1]).

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

There are two cases. When evaluating a call, the Erlang virtual machine (called BEAM) needs to determine which case has to be chosen.

This is decided by matching the call against the heads of the cases in turn, here `fact(0)` first then `fact(N)`, considered as patterns — assuming that a function name only matches the same name.

The first match selects the function body to be executed.

Matching/Calling functions (cont)

The procedure can be summarised as follows:

1. evaluate the arguments of the function call;
2. find the cases defining the called function;
3. try to match the head of the first case against the function call;
4. in case of success, the body of the case is evaluated with the bindings of the match;
5. otherwise the next case is tried;
6. if no case matches, an error is issued.

Matching/Calling functions (cont)

Consider a more complex case:

```
X=2, fact(X)
```

1. the argument `X` of `fact(X)` is evaluated with the bindings $\{\texttt{X} \mapsto 2\}$, so the call is now `fact(2)`;
2. the matching `fact(0) = fact(2)` fails,
3. so `fact(N) = fact(2)` is tried, and succeeds;
4. the expression `N * fact(N-1)` is evaluated with the bindings $\{\texttt{N} \mapsto 2\}$.

Matching/Calling functions (cont)

Consider a function definition relying on the matching of a list:

```
-module(list1).
-export([len/1]).

len([])     -> 0;
len([_|T]) -> 1 + len(T).
```

This function takes a list as argument and return the number of items in it.

There are two cases: one for the empty list and another for the non-empty list.