GUARDS IN ERLANG

Guards are additional clauses that can go in a function's head to make pattern matching more expressive. As mentioned above, pattern matching is somewhat limited as it cannot express things like a range of value or certain types of data. A concept we couldn't represent is counting: is this 12 years old basketball player too short to play with the pros? Is this distance too long to walk on your hands? Are you too old or too young to drive a car? You couldn't answer these with simple pattern matching. I mean, you could represent the driving question such as:

```
old_enough(0) -> false;
old_enough(1) -> false;
old_enough(2) -> false;
...
old_enough(14) -> false;
old_enough(15) -> false;
old_enough(_) -> true.
```

But it would be incredibly impractical. You can do it if you want, but you'll be alone to work on your code forever. If you want to eventually make friends, start a new `guards` module so we can type in the "correct" solution to the driving question:

```
old_enough(X) when X >= 16 -> true;
old_enough(_) -> false.
```

And you're done! As you can see, this is much shorter and cleaner. Note that a basic rule for guard expression is they must return `true` to succeed. The guard will fail if it returns `false` or if it throws an exception.

Suppose we now forbid people who are over 104 years old to drive. Our valid ages for drivers is now from 16 years old up to 104 years old. We need to take care of that, but how? Let's just add a second guard clause:

```
right_age(X) when X >= 16, X =< 104 ->
true;
right_age(_) ->
false.
```

The comma (`,`) acts in a similar manner to the operator `andalso` and the semicolon (`;`) acts a bit like `orelse`(described in "Starting Out (for real)"). Both guard expressions need to succeed for the whole guard to pass. We could also represent the function the opposite way:

```
wrong_age(X) when X < 16; X > 104 ->
```

```
true;
wrong_age(_) ->
false.
```

And we get correct results from that too. Test it if you want (you should always test stuff!). In guard expressions, the semi-colon (`;`) acts like the `orelse` operator: if the first guard fails, it then tries the second, and then the next one, until either one guard succeeds or they all fail.

You can use a few more functions than comparisons and boolean evaluation in functions, including math operations (`A*B/C >= 0`) and functions about data types, such as`is_integer/1`, `is_atom/1`, etc. (We'll get back on them in the following chapter). One negative point about guards is that they will not accept user-defined functions because of side effects. Erlang is not a purely functional programming language (like Haskell is) because it relies on side effects a lot: you can do I/O, send messages between actors or throw errors as you want and when you want. There is no trivial way to determine if a function you would use in a guard would or wouldn't print text or catch important errors every time it is tested over many function clauses. So instead, Erlang just doesn't trust you (and it may be right to do so!)

That being said, you should be good enough to understand the basic syntax of guards to understand them when you encounter them.

Note: I've compared `,` and `;` in guards to the operators `andalso` and `orelse`. They're not exactly the same, though. The former pair will catch exceptions as they happen while the latter won't. What this means is that if there is an error thrown in the first part of the guard `X >= N; N >= 0`, the second part can still be evaluated and the guard might succeed; if an error was thrown in the first part of `X >= N orelse N >= 0`, the second part will also be skipped and the whole guard will fail.

However (there is always a 'however'), only `andalso` and `orelse` can be nested inside guards. This means `(A orelse B) andalso C` is a valid guard, while `(A; B), C` is not. Given their different use, the best strategy is often to mix them as necessary.

Source : http://learnyousomeerlang.com/syntax-in-functions