For the calculation below, I prefer to use the more "mathematical" notation for regular expressions. Therefore let us first look at this notation and the corresponding Scala code.

"mathematical" notation for regular expressions	Scala code
$\mathbf{0}$	`ZERO`
$\mathbf{1}$	`ONE`
c	`CHAR(c)`
$\sum rs$	`ALTs(rs)`
$\prod rs$	`SEQs(rs)`
r^*	`STAR(r)`

My own convention is that `rs` stands for a list of regular expressions. Also of note is that these are **all** regular expressions in Main 3 and the template file defines them as the (algebraic) datatype `Rexp`. A confusion might arise from the fact that there is also some shorthand notation for some regular expressions, namely

```
def ALT(r1: Rexp, r2: Rexp) = ALTs(List(r1, r2))
def SEQ(r1: Rexp, r2: Rexp) = SEQs(List(r1, r2))
```

Since these are functions, everything of the form `ALT(r1, r2)` will immediately be translated into the regular expression `ALTs(List(r1, r2))` (similarly for `SEQ`). Maybe even more confusing is that Scala allows one to define *extensions* that provide an even shorter notation for `ALT` and `SEQ`, namely

$$\texttt{r1} \sim \texttt{r2} \stackrel{\text{def}}{=} \texttt{SEQ(r1, r2)} \stackrel{\text{def}}{=} \texttt{SEQs(List(r1, r2))}$$
$$\texttt{r1} \mid \texttt{r2} \stackrel{\text{def}}{=} \texttt{ALT(r1, r2)} \stackrel{\text{def}}{=} \texttt{ALTs(List(r1, r2))}$$

The right hand sides are the fully expanded definitions. The reason for this even shorter notation is that in the mathematical notation one often writes

$$r_1 \cdot r_2 \stackrel{\text{def}}{=} \prod [r_1, r_2]$$
$$r_1 + r_2 \stackrel{\text{def}}{=} \sum [r_1, r_2]$$

The first one is for binary *sequence* regular expressions and the second for binary *alternative* regular expressions. The regex in question in the shorthand notation is $(a + 1) \cdot a$, which is the same as

$$\prod [\Sigma [a, 1], a]$$

or in Scala code

$$\texttt{(CHAR('a') | ONE)} \sim \texttt{CHAR('a')}$$

Using the mathematical notation, the definition of *der* is given by the rules:

$$
\begin{array}{lll}
(1) & der\ c\ (\mathbf{0}) & \stackrel{\text{def}}{=} \mathbf{0} \\
(2) & der\ c\ (\mathbf{1}) & \stackrel{\text{def}}{=} \mathbf{0} \\
(3) & der\ c\ (d) & \stackrel{\text{def}}{=} if\ c = d\ then\ \mathbf{1}\ else\ \mathbf{0} \\
(4) & der\ c\ (\sum [r_1, .., r_n]) & \stackrel{\text{def}}{=} \sum [der\ c\ r_1, .., der\ c\ r_n] \\
(5) & der\ c\ (\prod []) & \stackrel{\text{def}}{=} \mathbf{0} \\
(6) & der\ c\ (\prod r :: rs) & \stackrel{\text{def}}{=} if\ nullable(r) \\
 & & \quad then\ (\prod (der\ c\ r) :: rs) + (der\ c\ (\prod rs)) \\
 & & \quad else\ (\prod (der\ c\ r) :: rs) \\
(7) & der\ c\ (r^*) & \stackrel{\text{def}}{=} (der\ c\ r) \cdot (r^*)
\end{array}
$$

Let's finally do the calculation for the derivative of the regular expression with respect to the letter a (in red is in each line which regular expression is analysed):

$$
\begin{array}{lll}
 & der(a, (a+1) \cdot a) & \text{by (6) and since } a+1 \text{ is nullable} \\
\stackrel{\text{def}}{=} & (der(a, a+1) \cdot a) + der(a, \prod [a]) & \text{by (4)} \\
\stackrel{\text{def}}{=} & ((der(a, a) + \mathtt{der}(a, \mathbf{1})) \cdot a) + der(a, \prod [a]) & \text{by (3)} \\
\stackrel{\text{def}}{=} & ((\mathbf{1} + \mathtt{der}(a, 1)) \cdot a) + der(a, \prod [a]) & \text{by (2)} \\
\stackrel{\text{def}}{=} & ((\mathbf{1} + \mathbf{0}) \cdot a) + der(a, \prod [a]) & \text{by (6) and } a \text{ not being nullable} \\
\stackrel{\text{def}}{=} & ((\mathbf{1} + \mathbf{0}) \cdot a) + \prod [\mathtt{der}(a, a)] & \text{by (3)} \\
\stackrel{\text{def}}{=} & ((\mathbf{1} + \mathbf{0}) \cdot a) + \prod [\mathbf{1}] &
\end{array}
$$

Translating this result back into Scala code gives you

```
ALT((ONE | ZERO) ~ CHAR('a'), SEQs(List(ONE)))
```