Handout 4 (Sulzmann & Lu Algorithm)

So far our algorithm based on derivatives is only able to say yes or no depending on whether a string is matched by a regular expression or not. Often a more interesting question is to find out *how* a regular expression matched a string? Answering this question will also help us with the problem we are after, namely tokenising an input string.

The algorithm we will be looking at in this lecture was designed by Sulzmann & Lu in a rather recent research paper (from 2014). A link to it is provided on KEATS, in case you are interested.[1] Together with my former PhD students Fahad Ausaf and Chengsong Tan we wrote several papers where we build on their result: we provided a mathematical proof that their algorithm is really correct—the proof Sulzmann & Lu had originally given contained major flaws. Such correctness proofs are important: Kuklewicz maintains a unit-test library for the kind of algorithms we are interested in here and he showed that many implementations in the "wild" are buggy, that is not satisfy his unit tests:

http://www.haskell.org/haskellwiki/Regex_Posix

Coming back to the algorithm by Sulzmann & Lu, their idea is to introduce *values* for producing an answer for *how* a regular expression matches a string. So rather than a boolean like in the Brzozowski algorithm, a value will be the output of the Sulzman & Lu algorithm whenever the regular expression matches the string. If the string does not match the string, an error will be raised. The definition for values is given below. They are shown together with the regular expressions r to which they correspond:

$$
\begin{array}{llll}
\text{regular expressions} & & \text{values} & \\
r \quad ::= \quad \mathbf{0} & & v \quad ::= & \\
\quad\mid\quad \mathbf{1} & & & \textit{Empty} \\
\quad\mid\quad c & & \mid & \textit{Char}(c) \\
\quad\mid\quad r_1 \cdot r_2 & & \mid & \textit{Seq}(v_1, v_2) \\
\quad\mid\quad r_1 + r_2 & & \mid & \textit{Left}(v) \\
& & \mid & \textit{Right}(v) \\
\quad\mid\quad r^* & & \mid & \textit{Stars}\,[v_1, \ldots\, v_n]
\end{array}
$$

There is no value for the **0** regular expression, since it does not match any string. Otherwise there is exactly one value corresponding to each regular expression with the exception of $r_1 + r_2$ where there are two values, namely *Left*(v) and *Right*(v) corresponding to the two alternatives. Note that r^* is associated with a list of values, one for each copy of r that was needed to match the string. This

---

[1]In my humble opinion this is an interesting instance of the research literature: it contains very clever ideas, but its presentation is rather sloppy. In earlier versions of this paper, students and I found several rather annoying typos in the examples and definitions; we even found downright errors in their work.
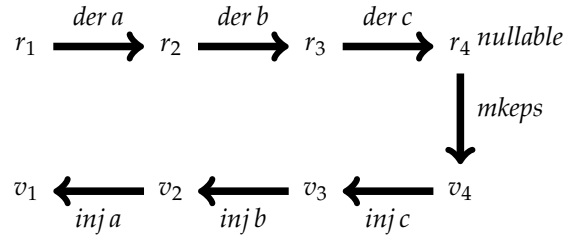
Figure 1: The two phases of the algorithm by Sulzmann & Lu.

means we might also return the empty list $Stars\,[]$, if no copy was needed for r^*. For sequence, there is exactly one value, composed of two component values (v_1 and v_2).

My implementation of regular expressions and values in Scala is shown below. I use the convention that regular expressions are written entirely with upper-case letters, whereas values start with a single upper-case character and the rest are lower-case letters.

```scala
abstract class Rexp
case object ZERO extends Rexp
case object ONE extends Rexp
case class CHAR(c: Char) extends Rexp
case class ALT(r1: Rexp, r2: Rexp) extends Rexp
case class SEQ(r1: Rexp, r2: Rexp) extends Rexp
case class STAR(r: Rexp) extends Rexp

abstract class Val
case object Empty extends Val
case class Chr(c: Char) extends Val
case class Sequ(v1: Val, v2: Val) extends Val
case class Left(v: Val) extends Val
case class Right(v: Val) extends Val
case class Stars(vs: List[Val]) extends Val
```

Graphically the algorithm by Sulzmann & Lu can be illustrated by the picture in Figure 1 where the path from the left to the right involving *der*/*nullable* is the first phase of the algorithm and *mkeps*/*inj*, the path from right to left, the second phase. This picture shows the steps required when a regular expression, say r_1, matches the string *abc*. We first build the three derivatives (according to *a*, *b* and *c*). We then use *nullable* to find out whether the resulting regular expression can match the empty string. If yes, we call the function *mkeps*. The *mkeps* function calculates a value for how a regular expression has matched the empty string. Its definition is as follows:

$$
\begin{aligned}
mkeps(\mathbf{1}) &\stackrel{\text{def}}{=} Empty \\
mkeps(r_1 + r_2) &\stackrel{\text{def}}{=} \text{if } nullable(r_1) \\
&\qquad \text{then } Left(mkeps(r_1)) \\
&\qquad \text{else } Right(mkeps(r_2)) \\
mkeps(r_1 \cdot r_2) &\stackrel{\text{def}}{=} Seq(mkeps(r_1), mkeps(r_2)) \\
mkeps(r^*) &\stackrel{\text{def}}{=} Stars\,[]
\end{aligned}
$$

There are no cases for $\mathbf{0}$ and c, since these regular expression cannot match the empty string. Note also that in case of alternatives we give preference to the regular expression on the left-hand side. This will become important later on.

The second phase of the algorithm is organised so that it will calculate a value for how the derivative regular expression has matched a string. For this we need a function that reverses this "chopping off" for values which we did in the first phase for derivatives. The corresponding function is called *inj* (short for injection). This function takes three arguments: the first one is a regular expression for which we want to calculate the value, the second is the character we want to inject and the third argument is the value where we will inject the character into. The result of this function is a new value. The definition of *inj* is as follows:

$$
\begin{aligned}
inj\,(c)\,c\,Empty &\stackrel{\text{def}}{=} Char\,c \\
inj\,(r_1 + r_2)\,c\,Left(v) &\stackrel{\text{def}}{=} Left(inj\,r_1\,c\,v) \\
inj\,(r_1 + r_2)\,c\,Right(v) &\stackrel{\text{def}}{=} Right(inj\,r_2\,c\,v) \\
inj\,(r_1 \cdot r_2)\,c\,Seq(v_1, v_2) &\stackrel{\text{def}}{=} Seq(inj\,r_1\,c\,v_1, v_2) \\
inj\,(r_1 \cdot r_2)\,c\,Left(Seq(v_1, v_2)) &\stackrel{\text{def}}{=} Seq(inj\,r_1\,c\,v_1, v_2) \\
inj\,(r_1 \cdot r_2)\,c\,Right(v) &\stackrel{\text{def}}{=} Seq(mkeps(r_1), inj\,r_2\,c\,v) \\
inj\,(r^*)\,c\,Seq(v, Stars\,vs) &\stackrel{\text{def}}{=} Stars(inj\,r\,c\,v :: vs)
\end{aligned}
$$

This definition is by recursion on the regular expression and by analysing the shape of the values. Therefore there are three cases for sequence regular expressions (for all possible shapes of the value we can encounter). The last clause for the star regular expression returns a list where the first element is $inj\,r\,c\,v$ and the other elements are vs. That means _ :: _ should be read as list cons.

To understand what is going on, it might be best to do some example calculations and compare them with Figure 1. For this note that we have not yet dealt with the need of simplifying regular expressions (this will be a topic on its own later). Suppose the regular expression is $a \cdot (b \cdot c)$ and the input string is abc. The derivatives from the first phase are as follows:

$$
\begin{aligned}
r_1: &\quad a \cdot (b \cdot c) \\
r_2: &\quad \mathbf{1} \cdot (b \cdot c) \\
r_3: &\quad (\mathbf{0} \cdot (b \cdot c)) + (\mathbf{1} \cdot c) \\
r_4: &\quad (\mathbf{0} \cdot (b \cdot c)) + ((\mathbf{0} \cdot c) + \mathbf{1})
\end{aligned}
$$

According to the simple algorithm, we would test whether r_4 is nullable, which in this case it indeed is. This means we can use the function *mkeps* to calculate a value for how r_4 was able to match the empty string. Remember that this function gives preference for alternatives on the left-hand side. However there is only **1** on the very right-hand side of r_4 (underlined)

$$r_4 \colon (\mathbf{0} \cdot (b \cdot c)) + ((\mathbf{0} \cdot c) + \underline{\mathbf{1}})$$

that matches the empty string. Therefore *mkeps* returns the value

$$v_4 \colon \; Right(Right(Empty))$$

If there had been a **1** on the left, then *mkeps* would have returned something of the form *Left*(...). The point is that from this value we can directly read off which part of r_4 matched the empty string: take the right-alternative first, and then the right-alternative again, then you got to the **1**.

Next we have to "inject" the last character, that is c in the running example, into this value v_4 in order to calculate how r_3 could have matched the string c. For this we call injection with $inj(r_3, c, v_4)$. According to the definition of *inj* we obtain

$$v_3 \colon \; Right(Seq(Empty, Char(c)))$$

This is the correct result, because r_3 needs to use the right-hand alternative, and then **1** needs to match the empty string and c needs to match c. Next we need to inject back the letter b into v_3. For this we call injection with $inj(r_2, b, v_3)$. This gives

$$v_2 \colon \; Seq(Empty, Seq(Char(b), Char(c)))$$

which is again the correct result for how r_2 matched the string bc. Finally we need to inject back the letter a into v_2 giving the final result. For this we call injection with $inj(r_1, a, v_2)$ and obtain

$$v_1 \colon \; Seq(Char(a), Seq(Char(b), Char(c)))$$

This value corresponds to how the regular expression r_1, namely $a \cdot (b \cdot c)$, matched the string abc.

There are a few auxiliary functions that are of interest when analysing this algorithm. One is called *flatten*, written $|\_|$, which extracts the string "underlying" a value. It is defined recursively as

$$
\begin{aligned}
|Empty| &\overset{\text{def}}{=} [] \\
|Char(c)| &\overset{\text{def}}{=} [c] \\
|Left(v)| &\overset{\text{def}}{=} |v| \\
|Right(v)| &\overset{\text{def}}{=} |v| \\
|Seq(v_1, v_2)| &\overset{\text{def}}{=} |v_1| \,@\, |v_2| \\
|Stars\,[v_1, \ldots, v_n]| &\overset{\text{def}}{=} |v_1| \,@\ldots@\, |v_n|
\end{aligned}
$$

Using flatten we can see what are the strings behind the values calculated by *mkeps* and *inj*. In our running example:

$$
\begin{array}{ll}
|v_4|: & [] \\
|v_3|: & c \\
|v_2|: & bc \\
|v_1|: & abc
\end{array}
$$

This indicates that *inj* indeed is injecting, or adding, back a character into the value.

The definition of *inj* might still be very puzzling and each clause might look arbitrary, but there is in fact a relatively simple idea behind it. Ultimately the *inj*-functions is determined by the derivative functions. For this consider one of the "squares" from Figure 1:
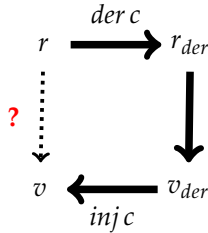
The input to the *inj*-function is r (on the top left), c (the character to be injected) and v_{der} (bottom right). The output is the value v for how the regular expression r matched the corresponding string. How does *inj* calculate this value? Well, it has to analyse the value v_{der} and transform it into the value v for the regular expression r. The value v_{der} corresponds to the r_{der}-regular expression which is the derivative of r. Remember that v_{der} is the value for how r_{der} matches the corresponding string where c has been chopped off.

For a concrete example, let r be $r_1 + r_2$. Then r_{der} is of the form $(der\, c\, r_1) + (der\, c\, r_2)$. What are the possible values corresponding to r_{der}? Well, they can be only of the form *Left*(...) and *Right*(...). Therefore you have two cases in the *inj* function – one for *Left* and one for *Right*.

$$
\begin{array}{l}
inj\,(r_1 + r_2)\, c\; Left(v) \;\stackrel{\text{def}}{=}\; \ldots \\
inj\,(r_1 + r_2)\, c\; Right(v) \stackrel{\text{def}}{=}\; \ldots
\end{array}
$$

Let's look next at the sequence case where $r = r_1 \cdot r_2$. What does the derivative of r look like? According to the definition it is:

$$
der\, c\, (r_1 \cdot r_2) \stackrel{\text{def}}{=} \text{if } nullable(r_1) \\
\text{then } (der\, c\, r_1) \cdot r_2 + der\, c\, r_2 \\
\text{else } (der\, c\, r_1) \cdot r_2
$$

As you can see there is a derivative in the if-branch and another in the else-branch. In the if-branch we have an alternative of the form $\_ + \_$. We already know what the possible values are for such a regular expression, namely *Left* or *Right*. Therefore we have in *inj* the two cases:

$$inj\,(r_1 \cdot r_2)\,c\;Left(Seq(v_1, v_2)) \stackrel{\text{def}}{=} \ldots$$
$$inj\,(r_1 \cdot r_2)\,c\;Right(v) \stackrel{\text{def}}{=} \ldots$$

In the first case we even know that it is not just a *Left*-value, but $Left(Seq(\ldots))$, because the corresponding regular expression in the derivation is $(der\,c\,r_1) \cdot r_2$. Hence we know it must be a *Seq*-value enclosed inside *Left*. The third clause for $r_1 \cdot r_2$ in the *inj*-function

$$inj\,(r_1 \cdot r_2)\,c\;Seq(v_1, v_2) \stackrel{\text{def}}{=} \ldots$$

corresponds to the else-branch in the derivative. In this case we know the derivative is of the form $(der\,c\,r_1) \cdot r_2$ and therefore the value must be of the form $Seq(\ldots)$.

Hopefully the *inj*-function makes now more sense. I let you explain the r^* case. What do the derivative of r^* and the corresponding value look like? Does this explain the shape of the clause?

$$inj\,(r^*)\,c\;Seq(v, Stars\,vs) \stackrel{\text{def}}{=} Stars(inj\,r\,c\,v\,::\,vs)$$

If yes, you made sense of the left-hand sides of the *inj*-definition.

How about the right-hand sides? Well, in the r^* case for example we have according to the square in the picture above a value v_{der} which says how the derivative r_{der} matched the string. Since the derivative is of the form $(der\,c\,r) \cdot (r^*)$ the corresponding value is of the form $Seq(v, Stars\,vs)$. But for r^* we are looking for a value for the original (top left) regular expression — so it cannot be a value of the shape $Seq(\ldots, Stars\ldots)$ because that is the shape for sequence regular expressions. What we need is a value of the form $Stars\ldots$ (remember the correspondence between values and regular expressions). This suggests to define the right hand side as

$$\ldots \stackrel{\text{def}}{=} Stars(inj\,r\,c\,v\,::\,vs)$$

It is a value of the right shape, namely *Stars*. It injected c into the first-value, which is in fact the value where we need in order to undo the derivative. Remember again the shape of the derivative of r^*. In place where we chopped off the c, we now need to do the *inj* of c. Therefore $inj\,r\,c\,v$ in the definition above. That is the same with the other clauses in *inj*.

Phew…Sweat…!#@☠%…Unfortunately, there is a gigantic problem with the described algorithm so far: it is very slow. To make it faster, we have to include in all this the simplification from Lecture 2…and what rotten luck: simplification messes things up and we need to *rectify* the mess. This is what we shall do next.

Simplification

Generally the matching algorithms based on derivatives do poorly unless the regular expressions are simplified after each derivative step. But this is a bit more involved in the algorithm of Sulzmann & Lu. So what follows might require you to read several times before it makes sense and also might require that you do some example calculations yourself. As a first example consider the last derivation step in our earlier example:

$$r_4 = der\, c\, r_3 = (\mathbf{0} \cdot (b \cdot c)) + ((\mathbf{0} \cdot c) + \mathbf{1}) \tag{1}$$

Simplifying this regular expression would just give us $\mathbf{1}$. Running *mkeps* with this $\mathbf{1}$ as input, however, would give us with the value *Empty* instead of

$$Right(Right(Empty))$$

that was obtained without the simplification. The problem is we need to recreate this more complicated value, rather than just return *Empty*. This will require what I call *rectification functions*. They need to be calculated whenever a regular expression gets simplified.

   Rectification functions take a value as argument and return a (rectified) value. In the example above the argument would be *Empty* and the output $Right(Right(Empty))$. Before we define these rectification functions in general, let us first take a look again at our simplification rules:

$$
\begin{aligned}
r \cdot \mathbf{0} &\;\mapsto\; \mathbf{0} \\
\mathbf{0} \cdot r &\;\mapsto\; \mathbf{0} \\
r \cdot \mathbf{1} &\;\mapsto\; r \\
\mathbf{1} \cdot r &\;\mapsto\; r \\
r + \mathbf{0} &\;\mapsto\; r \\
\mathbf{0} + r &\;\mapsto\; r \\
r + r &\;\mapsto\; r
\end{aligned}
$$

Applying them to r_4 in (1) will require several nested simplifications in order end up with just $\mathbf{1}$. However, it is possible to apply them in a depth-first, or inside-out, manner in order to calculate this simplified regular expression.

   The rectification we can implement by letting simp return not just a (simplified) regular expression, but also a rectification function. Let us consider the alternative case, $r_1 + r_2$, first. By going depth-first, we first simplify the component regular expressions r_1 and r_2. This will return simplified versions, say r_{1s} and r_{2s}, of the component regular expressions (if they can be simplified) but also two rectification functions f_{1s} and f_{2s}. We need to assemble them in order to obtain a rectified value for $r_1 + r_2$. In case r_{1s} simplified to $\mathbf{0}$, we continue the derivative calculation with r_{2s}. The Sulzmann & Lu algorithm would return a corresponding value, say v_{2s}. But now this value needs to be "rectified" to the value

$$Right(v_{2s})$$

The reason is that we look for the value that tells us how $r_1 + r_2$ could have matched the string, not just r_2 or r_{2s}. Unfortunately, this is still not the right value in general because there might be some simplifications that happened inside r_2 and for which the simplification function returned also a rectification function f_{2s}. So in fact we need to apply this one too which gives

$$Right(f_{2s}(v_{2s}))$$

This is now the correct, or rectified, value. Since the simplification will be done in the first phase of the algorithm, but the rectification needs to be done to the values in the second phase, it is advantageous to calculate the rectification as a function, remember this function and then apply the value to this function during the second phase. So if we want to implement the rectification as function, we would need to return

$$\lambda v. Right(f_{2s}(v))$$

which is the lambda-calculus notation for a function that expects a value v and returns everything after the dot where v is replaced by whatever value is given.

Let us package this idea with rectification functions into a single function (still only considering the alternative case):

```
1   simp(r):
2      case r = r₁ + r₂
3         let (r₁ₛ, f₁ₛ) = simp(r₁)
4             (r₂ₛ, f₂ₛ) = simp(r₂)
5         case r₁ₛ = 0: return (r₂ₛ, λv. Right(f₂ₛ(v)))
6         case r₂ₛ = 0: return (r₁ₛ, λv. Left(f₁ₛ(v)))
7         case r₁ₛ = r₂ₛ: return (r₁ₛ, λv. Left(f₁ₛ(v)))
8         otherwise: return (r₁ₛ + r₂ₛ, f_alt(f₁ₛ, f₂ₛ))
```

We first recursively call the simplification with r_1 and r_2 (Lines 3 and 4). This gives simplified regular expressions, r_{1s} and r_{2s}, as well as two rectification functions f_{1s} and f_{2s}. We next need to test whether the simplified regular expressions are $\mathbf{0}$ so as to make further simplifications. In case r_{1s} is $\mathbf{0}$ (Line 5), then we can return r_{2s} (the other alternative). However for this we need to build a corresponding rectification function, which as said above is

$$\lambda v. Right(f_{2s}(v))$$

The case where $r_{2s} = \mathbf{0}$ is similar: We return r_{1s} and rectify with $Left(\_)$ and the other calculated rectification function f_{1s}. This gives

$$\lambda v. Left(f_{1s}(v))$$

The next case where $r_{1s} = r_{2s}$ can be treated like the one where $r_{2s} = \mathbf{0}$. We return r_{1s} and rectify with $Left(\_)$ and so on.

The otherwise-case is slightly more complicated. In this case neither r_{1s} nor r_{2s} are $\mathbf{0}$ and also $r_{1s} \neq r_{2s}$, which means no further simplification can be applied. Accordingly, we return $r_{1s} + r_{2s}$ as the simplified regular expression. In principle we also do not have to do any rectification, because no simplification was done in this case. But this is actually not true: There might have been simplifications inside r_{1s} and r_{2s}. We therefore need to take into account the calculated rectification functions f_{1s} and f_{2s}. We can do this by defining a rectification function f_{alt} which takes two rectification functions as arguments and applies them according to whether the value is of the form $Left(\_)$ or $Right(\_)$:

$$f_{alt}(f_1, f_2) \stackrel{\text{def}}{=}$$
$$\lambda v. \ \text{case } v = Left(v'): \quad \text{return } Left(f_1(v'))$$
$$\text{case } v = Right(v'): \text{return } Right(f_2(v'))$$

The other interesting case with simplification is the sequence case. In this case the main simplification function is as follows

$$simp(r): \hspace{3cm} \text{(continued)}$$
$$\text{case } r = r_1 \cdot r_2$$
$$\text{let } (r_{1s}, f_{1s}) = simp(r_1)$$
$$(r_{2s}, f_{2s}) = simp(r_2)$$
$$\text{case } r_{1s} = \mathbf{0}: \text{return } (\mathbf{0}, f_{error})$$
$$\text{case } r_{2s} = \mathbf{0}: \text{return } (\mathbf{0}, f_{error})$$
$$\text{case } r_{1s} = \mathbf{1}: \text{return } (r_{2s}, \lambda v. \, Seq(f_{1s}(Empty), f_{2s}(v)))$$
$$\text{case } r_{2s} = \mathbf{1}: \text{return } (r_{1s}, \lambda v. \, Seq(f_{1s}(v), f_{2s}(Empty)))$$
$$\text{otherwise: return } (r_{1s} \cdot r_{2s}, f_{seq}(f_{1s}, f_{2s}))$$

whereby in the last line f_{seq} is again pushing the two rectification functions into the two components of the Seq-value:

$$f_{seq}(f_1, f_2) \stackrel{\text{def}}{=}$$
$$\lambda v. \ \text{case } v = Seq(v_1, v_2): \text{return } Seq(f_1(v_1), f_2(v_2))$$

Note that in the case of $r_{1s} = \mathbf{0}$ (similarly r_{2s}) we use the function f_{error} for rectification. If you think carefully, then you will realise that this function will actually never been called. This is because a sequence with $\mathbf{0}$ will never recognise any string and therefore the second phase of the algorithm would never been called. The simplification function still expects us to give a function. So in my own implementation I just returned a function that raises an error. In the case where $r_{1s} = \mathbf{1}$ (similarly r_{2s}) we have to create a sequence where the first component is a rectified version of $Empty$. Therefore we call f_{1s} with $Empty$.

Since we only simplify regular expressions of the form $r_1 + r_2$ and $r_1 \cdot r_2$ we do not have to do anything else in the remaining cases. The rectification function will be just the identity, which in lambda-calculus terms is just

$$\lambda v. \, v$$

$simp(r)$:
  case $r = r_1 + r_2$
    let $(r_{1s}, f_{1s}) = simp(r_1)$
      $(r_{2s}, f_{2s}) = simp(r_2)$

    case $r_{1s} = \mathbf{0}$: return $(r_{2s}, \lambda v.\, Right(f_{2s}(v)))$
    case $r_{2s} = \mathbf{0}$: return $(r_{1s}, \lambda v.\, Left(f_{1s}(v)))$
    case $r_{1s} = r_{2s}$: return $(r_{1s}, \lambda v.\, Left(f_{1s}(v)))$
    otherwise: return $(r_{1s} + r_{2s}, f_{alt}(f_{1s}, f_{2s}))$

  case $r = r_1 \cdot r_2$
    let $(r_{1s}, f_{1s}) = simp(r_1)$
      $(r_{2s}, f_{2s}) = simp(r_2)$

    case $r_{1s} = \mathbf{0}$: return $(\mathbf{0}, f_{error})$
    case $r_{2s} = \mathbf{0}$: return $(\mathbf{0}, f_{error})$
    case $r_{1s} = \mathbf{1}$: return $(r_{2s}, \lambda v.\, Seq(f_{1s}(Empty), f_{2s}(v)))$
    case $r_{2s} = \mathbf{1}$: return $(r_{1s}, \lambda v.\, Seq(f_{1s}(v), f_{2s}(Empty)))$
    otherwise: return $(r_{1s} \cdot r_{2s}, f_{seq}(f_{1s}, f_{2s}))$

  otherwise:
    return $(r, \lambda v.\, v)$

Figure 2: The simplification function that returns a simplified regular expression and a rectification function.

This completes the high-level version of the simplification function, which is shown again in Figure 2. The Scala code for the simplification function is in Figure 3.

We are now in the position to define a *lexing function* as follows:

$$lex\; r\; [] \quad \overset{\text{def}}{=} \quad \text{if } nullable(r) \text{ then } mkeps(r)$$
$$\text{else } error$$

$$lex\; r\; c::s \quad \overset{\text{def}}{=} \quad \text{let } (r_{simp}, f_{rect}) = simp(der(c, r))$$
$$inj\; r\; c\; f_{rect}(lex\; r_{simp}\; s)$$

This corresponds to the *matches* function we have seen in earlier lectures. In the first clause we are given an empty string, [], and need to test whether the regular expression is *nullable*. If yes, we can proceed normally and just return the value calculated by *mkeps*. The second clause is for strings where the first character is c, say, and the rest of the string is s. We first build the derivative of r with respect to c; simplify the resulting regular expression. We continue lexing with the simplified regular expression and the string s. Whatever will be returned as value, we still need to rectify using the f_{rect} from the simplification and finally inject c back into the (rectified) value.

```scala
def F_ID(v: Val): Val = v
def F_RIGHT(f: Val => Val) = (v:Val) => Right(f(v))
def F_LEFT(f: Val => Val) = (v:Val) => Left(f(v))
def F_ALT(f1: Val => Val, f2: Val => Val) = (v:Val) => v match {
  case Right(v) => Right(f2(v))
  case Left(v) => Left(f1(v))
}
def F_SEQ(f1: Val => Val, f2: Val => Val) = (v:Val) => v match {
  case Sequ(v1, v2) => Sequ(f1(v1), f2(v2))
}
def F_SEQ_Empty1(f1: Val => Val, f2: Val => Val) =
  (v:Val) => Sequ(f1(Empty), f2(v))
def F_SEQ_Empty2(f1: Val => Val, f2: Val => Val) =
  (v:Val) => Sequ(f1(v), f2(Empty))
def F_ERROR(v: Val): Val = throw new Exception("error")

// simplification of regular expressions returning also a
// rectification function; no simplification under STAR
def simp(r: Rexp): (Rexp, Val => Val) = r match {
  case ALT(r1, r2) => {
    val (r1s, f1s) = simp(r1)
    val (r2s, f2s) = simp(r2)
    (r1s, r2s) match {
      case (ZERO, _) => (r2s, F_RIGHT(f2s))
      case (_, ZERO) => (r1s, F_LEFT(f1s))
      case _ => if (r1s == r2s) (r1s, F_LEFT(f1s))
                else (ALT (r1s, r2s), F_ALT(f1s, f2s))
    }
  }
  case SEQ(r1, r2) => {
    val (r1s, f1s) = simp(r1)
    val (r2s, f2s) = simp(r2)
    (r1s, r2s) match {
      case (ZERO, _) => (ZERO, F_ERROR)
      case (_, ZERO) => (ZERO, F_ERROR)
      case (ONE, _) => (r2s, F_SEQ_Empty1(f1s, f2s))
      case (_, ONE) => (r1s, F_SEQ_Empty2(f1s, f2s))
      case _ => (SEQ(r1s,r2s), F_SEQ(f1s, f2s))
    }
  }
  case r => (r, F_ID)
}
```

Figure 3: The Scala code for the simplification function. The first part defines some auxiliary functions for the rectification. The second part give the simplification function.

Records

Remember we wanted to tokenize input strings, that means splitting strings into their "word" components. Furthermore we want to classify each token as being a keyword or identifier and so on. For this one more feature will be required, which I call a *record* regular expression. While values encode how a regular expression matches a string, records can be used to "focus" on some particular parts of the regular expression and "forget" about others.

Let us look at an example. Suppose you have the regular expression $a \cdot b + a \cdot c$. Clearly this regular expression can only recognise two strings. But suppose you are not interested whether it can recognise ab or ac, but rather if it matched, then what was the last character of the matched string…either b or c. You can do this by annotating the regular expression with a record, written in general $(x : r)$, where x is just an identifier (in my implementation a plain string) and r is a regular expression. A record will be regarded as a regular expression. The extended definition in Scala therefore looks as follows:

```
abstract class Rexp
case object ZERO extends Rexp
case object ONE extends Rexp
case class CHAR(c: Char) extends Rexp
case class ALT(r1: Rexp, r2: Rexp) extends Rexp
case class SEQ(r1: Rexp, r2: Rexp) extends Rexp
case class STAR(r: Rexp) extends Rexp
case class REC(x: String, r: Rexp) extends Rexp
```

Since we regard records as regular expressions we need to extend the functions *nullable* and *der*. Similarly *mkeps* and *inj* need to be extended. This means we also need to extend the definition of values, which in Scala looks as follows:

```
abstract class Val
case object Empty extends Val
case class Chr(c: Char) extends Val
case class Sequ(v1: Val, v2: Val) extends Val
case class Left(v: Val) extends Val
case class Right(v: Val) extends Val
case class Stars(vs: List[Val]) extends Val
case class Rec(x: String, v: Val) extends Val
```

Let us now look at the purpose of records more closely and let us return to our question whether the string terminated in a b or c. We can do this as follows: we annotate the regular expression $ab + ac$ with a record as follows

$$a \cdot (x : b) + a \cdot (x : c)$$

This regular expression can still only recognise the strings ab and ac, but we can now use a function that takes a value and returns all records. I call this function *env* for environment…it builds a list of identifiers associated with a string. This function can be defined as follows:

$$
\begin{aligned}
env(Empty) &\overset{\text{def}}{=} [] \\
env(Char(c)) &\overset{\text{def}}{=} [] \\
env(Left(v)) &\overset{\text{def}}{=} env(v) \\
env(Right(v)) &\overset{\text{def}}{=} env(v) \\
env(Seq(v_1, v_2)) &\overset{\text{def}}{=} env(v_1) @ env(v_2) \\
env(Stars\,[v_1, \ldots, v_n]) &\overset{\text{def}}{=} env(v_1) @ \ldots @ env(v_n) \\
env(Rec(x : v)) &\overset{\text{def}}{=} (x : |v|) :: env(v)
\end{aligned}
$$

where in the last clause we use the flatten function defined earlier. As can be seen, the function *env* "picks" out all underlying strings where a record is given. Since there can be more than one, the environment will potentially contain many "records". If we now postprocess the value calculated by *lex* extracting all records using *env*, we can answer the question whether the last element in the string was an *b* or a *c*. Lets see this in action: if we use $a \cdot b + a \cdot c$ and *ac* the calculated value will be

$$Right(Seq(Char(a), Char(c)))$$

If we use instead $a \cdot (x : b) + a \cdot (x : c)$ and use the *env* function to extract the recording for *x* we obtain

$$[(x : c)]$$

If we had given the string *ab* instead, then the record would have been $[(x : b)]$. The fun starts if we iterate this. Consider the regular expression

$$(a \cdot (x : b) + a \cdot (y : c))^*$$

and the string *ababacabacab*. This string is clearly matched by the regular expression, but we are only interested in the sequence of *b*s and *c*s. Using *env* we obtain

$$[(x : b), (x : b), (y : c), (x : b), (y : c), (x : b)]$$

While this feature might look silly, it is in fact quite useful. For example if we want to match the name of an email we might use the regular expression

$$(name : [a\text{-}z0\text{-}9\_.\text{-}]^+) \cdot @ \cdot (domain : [a\text{-}z0\text{-}9\,\text{-}]^+) \cdot \ldots \cdot (top\_level : [a\text{-}z\,.]^{\{2,6\}})$$

Then if we match the email address

```
christian.urban@kcl.ac.uk
```

we can use the *env* function and find out what the name, domain and top-level part of the email address are:

```
write "Fib";
read n;
minus1 := 0;
minus2 := 1;
while n > 0 do {
        temp := minus2;
        minus2 := minus1 + minus2;
        minus1 := temp;
        n := n - 1
};
write "Result";
write minus2
```

Figure 4: The Fibonacci program in the While-language.

$$[(name : \texttt{christian.urban}), (domain : \texttt{kcl}), (top\_level : \texttt{ac.uk})]$$

Recall that we want to lex a little programming language, called the *While*-language. A simple program in this language is shown in Figure 4. The main keywords in this language are while, if, then and else.[2] As usual we have syntactic categories for identifiers, operators, numbers and so on. For this we would need to design the corresponding regular expressions to recognise these syntactic categories. I let you do this design task. Having these regular expressions at our disposal, we can form the regular expression

$$WhileRegs \stackrel{\text{def}}{=} \left(\begin{array}{l} (k, KeyWords) + \\ (i, Ids) + \\ (o, Ops) + \\ (n, Nums) + \\ (s, Semis) + \\ (p, (LParens + RParens)) + \\ (b, (Begin + End)) + \\ (w, WhiteSpaces) \end{array} \right)^{*}$$

and ask the algorithm by Sulzmann & Lu to lex, say the following string

```
"if true then then 42 else +"
```

---

[2]Contrast this with the COBOL programming language, which was developed around 1960 and thought to be dead for many decades—even your friendly lecturer is not old enough to have been taught this language. Anyway, this language had over 600 keywords (or what they called *reserved words*). Interestingly though this language is still used in 2020: during the height of Corona crisis the State of New Jewrsey in the US was looking for COBOL programers who could fix the state's national insurance webpage. You were probably paid in gold and diamonds, if you were able to program in COBOL. If you fixed their webpage, surely you were allowed to marry the governer's son/daughter.

By using the records and extracting the environment, the result is the following list:

```
(k, if),
(w, " "),
(i, true),
(w, " "),
(k, then),
(w, " "),
(k, then),
(w, " "),
(n, 42),
(w, " "),
(k, else),
(w, " "),
(o, +)
```

Typically we are not interested in the whitespaces and comments and would filter them out: this gives

```
(k, if),
(i, true),
(k, then),
(k, then),
(n, 42),
(k, else),
(o, +)
```

which will be the input for the next phase of our compiler.