1 Regular Expressions

This assignment will be assessed to determine 50% of your final mark. Please complete the indicated tasks and write a brief document explaining your work. You may prepare this document using Isabelle’s theory presentation facility, but this is not required. (A very simple way to print a theory file legibly is to use the Proof General command Isabelle > Commands > Display draft. You can combine the resulting output with a document produced using your favourite word processing package.) A clear write-up describing elegant, clearly structured proofs of all tasks will receive maximum credit.

You must work on this assignment as an individual. Collaboration is not permitted.

Consider reading, e.g., http://en.wikipedia.org/wiki/Regular_expression to refresh your knowledge of regular expressions.

For this assignment, we define regular expressions (over an arbitrary type 'a of characters) as follows:

1. ∅ is a regular expression.
2. ε is a regular expression.
3. If c is of type 'a, then Atom(c) is a regular expression.
4. If x and y are regular expressions, then xy is a regular expression.
5. If x and y are regular expressions, then x + y is a regular expression.
6. If x is a regular expression, then x* is a regular expression.

Nothing else is a regular expression.
Define a corresponding Isabelle/HOL data type. (Your concrete syntax may be different from that used above. For instance, you could write Star \( x \) for \( x^* \).)

\[
\text{datatype } 'a \text{ regexp} = \text{EmptySet} ("\emptyset")
\mid \text{EmptyWord} ("\epsilon")
\mid \text{Atom } 'a
\mid \text{Seq } "'a \text{ regexp}" "'a \text{ regexp}" \ (\text{infixl } "." 70)
\mid \text{Sum } "'a \text{ regexp}" "'a \text{ regexp}" \ (\text{infixl } "+" 65)
\mid \text{Star } "'a \text{ regexp}" \ ("_\ast" [80] 80)
\]

1.1 Regular Languages

A word is a list of characters:

\[
\text{type_synonym } 'a \text{ word} = "'a \text{ list}"
\]

Regular expressions denote formal languages, i.e., sets of words. For \( x \) a regular expression, we define its language \( L(x) \) as follows:

1. \( L(\emptyset) = \emptyset \).
2. \( L(\epsilon) = \{[]\} \).
3. \( L(\text{Atom}(c)) = \{[c]\} \).
4. \( L(xy) = \{uv \mid u \in L(x) \land v \in L(y)\} \).
5. \( L(x+y) = L(x) \cup L(y) \).
6. \( L(x^*) \) is the smallest set that contains the empty word and is closed under concatenation with words in \( L(x) \). That is, (i) \( [] \in L(x^*) \), and (ii) if \( u \in L(x) \) and \( v \in L(x^*) \), then \( uv \in L(x^*) \).

Define a function \( L \) that maps regular expressions to their language.

\[
\text{fun } L \ : \ "'a \text{ regexp} \Rightarrow 'a \text{ word set}" \text{ where}
\]
\[
\"L \emptyset = \{\}\"
\mid \"L \epsilon = \{[]\}\"
\mid \"L \text{ (Atom } c) = \{[c]\}\"
\]

\[
\text{inductive set } \text{KleeneStar} :: 'a \text{ word set} \Rightarrow 'a \text{ word set}
\text{ for } x :: "'a \text{ word set}" \text{ where}
\text{KleeneStar\_epsilon \ [simp]: } \[] \in \text{KleeneStar } x"
\mid \text{KleeneStar\_step: } \[ u \in x; v \in \text{KleeneStar } x \] \Rightarrow u \otimes v \in \text{KleeneStar } x"
\]

2
Prove the following lemma.

**lemma** KleeneStar_monotone [simp]: "u ∈ x ⇒ u ∈ KleeneStar x"
by (metis appendNIL2 KleeneStar_epsilon KleeneStar_step)

**lemma** KleeneStar_append [simp]:
"[ u ∈ KleeneStar x; v ∈ KleeneStar x ] ⇒ u @ v ∈ KleeneStar x"
by (induct u rule: KleeneStar.induct) (simp, simp add: KleeneStar_step)

**lemma** KleeneStar_idem:
"u ∈ KleeneStar (KleeneStar x) ⇒ u ∈ KleeneStar x"
by (induct u rule: KleeneStar.induct) simp_all

**lemma** "L (Star (Star x)) = L (Star x)"
by auto (erule KleeneStar_idem)

### 1.2 Matching via Derivatives

We now consider regular expression matching: the problem of determining whether a given word is in the language of a given regular expression. You are about to develop your own verified regular expression matcher. We need some auxiliary notions first.

A regular expression is called **nullable** iff its language contains the empty word.

▷ Define a recursive function **nullable** x that computes (by recursion over x, i.e., without explicit use of L) whether a regular expression is nullable.

**fun** nullable :: "'a regexp ⇒ bool" where
  "nullable ∅ = False"
/ "nullable ε = True"
/ "nullable (Atom c) = False"
/ "nullable (x · y) = (nullable x ∧ nullable y)"
/ "nullable (x+y) = (nullable x ∨ nullable y)"
/ "nullable (x*) = True"

▷ Prove the following lemma.

**lemma** "nullable x = ([ ] ∈ L x)"
by (induct x) auto
The derivative of a language $L$ with respect to a word $u$ is defined to be $\delta_u L = \{v \mid uv \in L\}$.

For languages that are given by regular expressions, there is a natural algorithm to compute the derivative as another regular expression.

Define a recursive function $\Delta c x$ that computes (by recursion over $x$) a regular expression whose language is the derivative of $L x$ with respect to the single-character word $[c]$.

```haskell
fun \(\Delta\) :: 'a ⇒ 'a regexp ⇒ 'a regexp where
  "\(\Delta\) c ∅ = ∅"
/ "\(\Delta\) c ε = ∅"
/ "\(\Delta\) c (Atom a) = (if c = a then ε else ∅)"
/ "\(\Delta\) c (x·y) = \(\Delta\) c x · y + (if nullable x then \(\Delta\) c y else ∅)"
/ "\(\Delta\) c (x+y) = \(\Delta\) c x + \(\Delta\) c y"
/ "\(\Delta\) c (x*) = \(\Delta\) c x · x*"
```

Hint: `nullable` might come in handy.

Prove the following lemma.

```isar
lemma KleeneStar_append_Cons [simp]:
  "[ c # u ∈ KleeneStar x; v ∈ KleeneStar x ] \implies c # u @ v ∈ KleeneStar x"
by (metis KleeneStar_append append_Cons)
```

```isar
lemma KleeneStar_split_nonempty:
  "c # w ∈ KleeneStar x \implies \exists u v. w = u @ v ∧ c # u ∈ x ∧ v ∈ KleeneStar x"
by (induct "c # w" rule: KleeneStar.induct) (auto simp add: append_eq_Cons_conv)
```

Alternatively, we can introduce a fresh variable as an abbreviation for the term $c # w$ that we want to induct over:

```isar
lemma "y ∈ KleeneStar x \implies y = c # w \implies \exists u v. w = u @ v ∧ c # u ∈ x ∧ v ∈ KleeneStar x"
by (induct y rule: KleeneStar.induct) (auto simp add: append_eq_Cons_conv)
```

```isar
lemma "u ∈ L (\(\Delta\) c x) = (c#u ∈ L x)"
proof (induct x arbitrary: u)
  case Seq thus ?case
    by (auto simp add: nullable_correct) (metis append_Cons, metis append_eq_Cons_conv)+
  case Star thus ?case
    by (auto simp add: KleeneStar_split_nonempty)
qed simp_all — the remaining cases are solved by simplification
```

Hint: see the Tutorial on Isabelle/HOL and the Tutorial on Isar for advanced induction
Define a recursive function $\delta$ that lifts $\Delta$ from single characters to words, i.e., $\delta \ u \ x$ is a regular expression whose language is the derivative of $L \ x$ with respect to the word $u$.

```isabelle
fun $\delta$ :: "'a word $\Rightarrow$ 'a regexp $\Rightarrow$ 'a regexp" where  "$\delta \ [] \ x = x"  "$\delta \ (c#cs) \ x = \delta \ cs \ (\Delta \ c \ x)"
```

Prove the following lemma.

```isabelle
lemma "u $\in$ L ($\delta \ v \ x) = (v @ u \in L \ x)"  by (induct v arbitrary: x) (simp, simp add: Delta_correct)
```

To obtain a regular expression matcher, we finally observe that $u \in L \ x$ if and only if $\delta \ u \ x$ is nullable.

```isabelle
definition match :: "'a word $\Rightarrow$ 'a regexp $\Rightarrow$ bool" where  "match u x = nullable (\delta \ u \ x)"
```

Prove correctness of `match`.

```isabelle
theorem "match u x = (u \in L \ x)"  by (simp add: match_def nullable_correct delta_correct)
```

▷ Solutions are due on Friday, June 17, 2011, at 12 noon. Please deliver a printed copy of the completed assignment to student administration by that deadline, and also send the corresponding Isabelle theory file to tw333@cam.ac.uk.