1 Background

In this lecture we introduced Regular Expressions which define a particularly simple class of languages, creatively called the class of Regular Languages (recall a language is a set of strings).

The standard way of checking if a string is among a set of strings $L$ is by defining a Turing Machine $M_L$ which accepts a string if it is in $L$, and rejects otherwise. If such a $M_L$ exists, $L$ is called Decidable.

While the subject of decidable languages and turing machines is beyond the scope of this class (it is primarily the topic of classes such as 15-251) we will include a brief description here for the sake of completeness.

A Turing Machine $M$ is (informally)\(^1\) a machine which can be in one of finitely many states, and which has access to an infinite tape which it can read from and write to. There are two special states, called ACCEPT and REJECT. The turing machine halts when it reaches one of these two states.

We say that a turing machine accepts a string if, when the machine is initialized to have that string at the beginning of the tape, it enters the ACCEPT state after finitely many steps.

Notice, however, that these machines might infinite loop, that is, they might never reach either of the ACCEPT or REJECT states. We are interested in a simpler class of turing machines, called Deterministic Finite Automata or DFAs.

A DFA is a turing machine which only moves to the right, and which is guaranteed to either accept or reject upon reaching the end of the input string. If a language $L$ can be decided by a DFA (that is to say, $s \in L$ iff the DFA accepts $s$), we call $L$ a Regular Language.

2 Regular Expressions

A theorem due to Kleene and Thompson shows that there is a nice recursive structure to regular languages.\(^2\) In particular, every regular language can be constructed in the following way:

\[ \emptyset \]

\[ \{ \text{""} \} \]

for all characters $c$ \[ \{ \text{"c"} \} \]

if $L_1$ and $L_2$ regular \[ L_1 \cup L_2 \]

if $L_1$ and $L_2$ regular \[ L_1 L_2 = \{l_1 l_2 | l_1 \in L_1, l_2 \in L_2 \} \]

if $L$ regular \[ L^* = \{l_1 l_2 \ldots l_n | l_i \in L \} \]

\(^1\)For a more formal treatment, see https://bit.ly/2sTGN7s and https://bit.ly/1Q7t6zt

\(^2\)Unfortunately this theorem is well beyond the scope of this class. If you’re at all interested in this sort of material, take Computational Discrete Math (15-354).
It does not take much work to show that the collection of all regular languages admits a sort of arithmetic in which $\emptyset$ acts like 0, \{"\}\ acts like 1, $L_1 \cup L_2$ acts like $L_1 + L_2$, and $L_1 L_2$ acts like $L_1 \times L_2$. To that end, we recursively define **Regular Expressions** as:

\[
\begin{align*}
0 \\
1 \\
c \\
R_1 + R_2 \\
R_1 \times R_2 \\
R_1^*
\end{align*}
\]

Where $R_1$ and $R_2$ are preexisting regular expressions. Note we will often write $R_1 \times R_2$ as $R_1 R_2$, as we do in standard arithmetic.

We additionally define the **Language** of a regular expression, $L(R)$ to be:

\[
\begin{align*}
L(0) &= \emptyset \\
L(1) &= \{"\}\ \\
L(c) &= \{"c"\} \\
L(R_1 + R_2) &= L(R_1) \cup L(R_2) \\
L(R_1 \times R_2) &= L(R_1) L(R_2) \\
L(R^*) &= 1 + L(R) L(R^*)
\end{align*}
\]

Exercises:

- Check that this definition makes sense given the recursive definition of regular languages above. In particular understand the $L(R^*)$ case.
- Understand why $L(aa) = \{"aa"\}$
- Understand why $L(a + b) = \{"a", "b"\}$
- Understand why $L(a^*) = \{"a", "aa", "aaa" \ldots\}$
- Understand why $L(ab^*) = \{"a", "ab", "abb", "abbb" \ldots\}$
- Understand why $L((ab)^*) = \{"a", "ab", "abab" \ldots\}$
- Understand why $\forall R. L(0 + R) = L(R + 0) = L(R)$
- Understand why $\forall R. L(1 R) = L(R 1) = L(R)$
- Understand why $L((a + b)^*) = \{\text{all finite strings containing only } a \text{ and } b\}$
- Write a regular expression $R$ such that $L(R) = \{\text{all finite strings of } a \text{ and } b \text{ containing exactly } 2 \text{ } a\}$
- Prove or disprove: $\forall R_1, R_2. L(R_1 + R_2) = L(R_2 + R_1)$
- Prove or disprove: $\forall R_1, R_2. L(R_1 R_2) = L(R_2 R_1)$

3 SML Implementation

Now that we have an understanding of regular expressions, we might want to write programs which use them. For instance, the `find` feature, bound to either ctrl-f or cmd-f depending on your operating system works by using regular expressions. Vim and Emacs both have features to find regular expressions in a document that is being edited, and the entire programming language of Perl was initially created (in part) to simplify the manipulation of regular expressions. Let’s implement some basic functionality in SML!

First, we need a datatype to represent our regular expressions. Since the mathematical definition has a nice recursive structure, it is very easy for us to immitate it exactly in SML:

\[\text{In particular, it exhibits a ring structure}\]
Datatype in hand, let’s write the function \( \text{accepts} : \text{regex} \to \text{string} \to \text{bool} \) with 

\[
\text{accepts } r \ s \iff s \in \mathcal{L}(r).
\]

Looking ahead, when we implement the \( \text{Times } (r_1,r_2) \) case, we will need to split our string into two parts, with the first part in the language of \( r_1 \) and the second part in the language of \( r_2 \). To facilitate this, we will use a helper function which handles this splitting elegantly. Inspired by CPS, we will write a function \( \text{matches} : \text{regex} \to \text{char list} \to (\text{char list} \to \text{bool}) \to \text{bool} \) which modifies its input function in order to keep track of the splitting that we do in the \( \text{Times} \) case. We will want \( \text{matches} \) to satisfy the following spec:

\[
\forall r : \text{regex} \\
\forall cs : \text{char list} \\
\forall k : \text{char list} \to \text{bool} \\
\text{matches } r \ cs \ k \iff \begin{cases} 
\text{cs} = \text{cs}_1@\text{cs}_2 \\
\text{cs}_1 \in \mathcal{L}(r) \\
k \ \text{cs}_2
\end{cases}
\]

Notice that \( \text{matches} \) splits \( cs \) in two, checking that the first half is in \( \mathcal{L}(r) \) and the second half satisfies \( k \). This sounds exactly like the \( \text{Times } (r_1,r_2) \) case, and you can see in the following implementation that we leverage this similarity (and in fact, we created this helper precisely for that similarity)

\[
\begin{align*}
\text{fun matches } ZERO \ cs \ k & = \text{false} \\
\text{matches ONE } cs \ k & = k \ cs \\
\text{matches } (\text{Char } c) \ cs \ k & = \\
\text{\hspace{1cm}} \text{case } cs \text{ of } [] \Rightarrow \text{false} \ | \ x::xs \Rightarrow (x=c) \text{ andalso } k \ xs \\
\text{matches } (\text{Plus } (r_1,r_2)) \ cs \ k & = \\
\text{\hspace{1cm}} (\text{matches } r_1 \ cs \ k) \text{ orelse } (\text{matches } r_2 \ cs \ k) \\
\text{matches } (\text{Times } (r_1,r_2)) \ cs \ k & = \\
\text{\hspace{1cm}} \text{matches } r_1 \ cs \ (\text{fn } cs_2 \Rightarrow \text{matches } r_2 \ cs_2 \ k) \\
\text{matches } (\text{Star } r) \ cs \ k & = \\
\text{\hspace{1cm}} \text{let } \\
\text{\hspace{2cm}} \text{fun } k' \ cs_2 \Rightarrow (cs_2 <> cs) \text{ andalso } (\text{matches } (\text{Star } r) \ cs_2 \ k) \text{ in} \\
\text{\hspace{2cm}} (k \ cs) \text{ orelse } (\text{matches } r \ cs \ k') \\
\text{end}
\end{align*}
\]

Notice that we can write \( \text{accepts} \) in terms of \( \text{matches} \):

\[
\text{fun accepts } r \ s = \text{matches } r \ (\text{String.explode } s) \ (\text{fn } cs_2 \Rightarrow cs_2 = [])
\]

Further, if \( \text{matches} \) is correct, then so is \( \text{accepts} \). (Exercise: prove this)
4 The Proof™

Theorem: matches satisfies its spec.

We proceed by structural induction on regex.

Case 1: ZERO

matches ZERO cs k = false

matches ZERO cs k should return true iff cs = cs1@cs2 with cs1 ∈ L(ZERO) = ∅ 
Since this can’t happen, we should always return false, and we do.

Case 2: ONE

matches ONE cs k = k cs

matches ONE cs k should return true iff cs = cs1@cs2 with
1. cs1 ∈ L(ONE) = {"c"}
2. k cs2 = true
Then cs1 = [] (by 1), and so cs2 = cs.
Thus we should return true iff k cs2 = true (by 2), which we do.

Case 3: Char c

matches (Char c) cs k = case cs of [] => false | x::xs => (x=c) andalso k xs

matches (Char c) cs k should return true iff cs = cs1@cs2 with
1. cs1 ∈ L(Char c) = {"c"}
2. k cs2 = true
This (in particular 1) is impossible if cs = [], and the code reflects this.
If cs = x::xs then cs1 = [x] and cs2 = xs
Then we should accept if
1. [x] ∈ {"c"} (namely if x=c)
2. k xs = true
Which is precisely what the code does.

Case 4: Plus (r1,r2)

matches (Plus (r1,r2)) cs k = (matches r1 cs k) orelse (matches r2 cs k)

matches (Plus (r1,r2)) cs k should return true iff cs = cs1@cs2 with
1. cs1 ∈ L(Plus (r1,r2)) = L(r1) ∪ L(r2)
2. k cs2 = true
Notice that this is true iff cs1 ∈ L(r1) or cs1 ∈ L(r2) with k cs2 = true
By induction, this is exactly what our recursive calls give us.

Case 5: Times (r1,r2)

matches (Times (r1,r2)) cs k = matches r1 cs (fn cs2 => matches r2 cs2 k)

matches (Times (r1,r2)) cs k should return true iff cs = cs1@cs2 with
1. cs1 ∈ L(Times (r1,r2)) = L(r1)L(r2)
2. k cs2 = true
By induction we return true iff cs = cs1@cs2 with
cs1 ∈ L(r1)
matches r2 cs2 k = true
By induction again, the second clause above is true iff cs2 = cs3@cs4 with
cs3 ∈ L(r2)
k cs4 = true
Note that if cs1 ∈ L(r1) and cs3 ∈ L(r2) then cs1@cs3 ∈ L(r1)L(r2)
Finally, to summarize, we return \( \text{true} \) iff \( cs = cs1@cs2 = cs1@cs3@cs4 \) with
- \( cs1 \in \mathcal{L}(r1) \)
- \( cs3 \in \mathcal{L}(r2) \)
- \( k \ cs4 = \text{true} \)

which is true iff
- \( cs1@cs3 \in \mathcal{L}(\text{Times}(r1,r2)) \)
- \( k \ cs4 = \text{true} \)

as desired.

**Case 6: Star r**

```
matches (Star r) cs k =
let
  fun k' cs2 = (cs2 <> cs) andalso (matches (Star r) cs2 k)
in
  (k cs) orelse (matches r cs k')
end
```

matches (Star r) cs k should return \( \text{true} \) iff \( cs = cs1@cs2 \) with
1. \( cs1 \in \mathcal{L}(\text{Star r}) \)
   \( = \mathcal{L}(\text{Plus}(\text{ONE},\text{Times}(r,\text{Star r}))) \)
   \( = \mathcal{L}(\text{ONE}) \cup \mathcal{L}(r)\mathcal{L}(\text{Star r}) \)
   \( = \{\epsilon\} \cup \mathcal{L}(r)\mathcal{L}(\text{Star r}) \)
2. \( k \ cs2 = \text{true} \)

We will also induct on the length of \( cs \)
If \( cs = [\] \), then we have \( cs = cs1 = cs2 = [\] \).
By definition, \( cs1 = [\] \in \mathcal{L}(\text{Star r}) \), so we should return \( k \ cs2 = k [\] = k cs \).
By induction on regex, the recursive call to matches returns \( \text{false} \)
(because \( k' \) gets called on \( cs2 = [\] = cs \) and so returns \( \text{false} \))
Thus we return \( \text{true} \) exactly when \( k \ cs = \text{true} \), as desired.

If \( cs = x::xs \), then:
If \( cs1 = [\] \) and \( cs2 = cs \), we return \( \text{true} \) exactly when \( k \ cs = \text{true} \), as desired.
(By similar logic to the base case)
Alternatively, if \( cs1 <> [\] \),
By induction on regex, \( \text{matches r cs k'} \) returns \( \text{true} \) iff
- \( cs1 \in \mathcal{L}(r) \)
- \( k' \ cs2 = \text{true} \)

But since \( cs1 <> [\] \) and \( cs1@cs2 = cs, cs2 <> cs \), and has strictly smaller length.
Then the first clause of \( k' \) is satisfied since \( cs2 <> cs \)
And the call to \( \text{matches (Star r) cs2 k} \) is correct by induction on the length of \( cs \).
So by similar logic to the \text{Times} case above, our original call is correct.