Lesson 13
REGULAR EXPRESSIONS

July 3, 2023

Lesson Plan

λ

1 Text Validation

2 Regular Expressions

3 Finite-State Automata

4 Matching Regular Expressions

5 Proving Correctness

1 - Text Validation

Suppose that we are interested in the problem of validating user input.

In this particular circumstance, assume that we are taking input that is supposed to look like an email. We expect an input which looks something like

$$\langle name \rangle @ \langle website \rangle . \langle extension \rangle$$

```
validateEmail : string -> bool
```
REQUIRES: `true`
ENSURES: `validateEmail s` $\Longrightarrow^*$ `true` iff `s` is a valid email address

We note that this problem can be solved by noticing what characters fall into each section of the email. We know that `@` and `.` need to show up, but there are different requirements for the name, website, and extension!

For simplicity, we assume that:

- an $\langle extension \rangle$ can only be "org" or "com"
- a $\langle website \rangle$ must be fully alphanumeric
- a $\langle name \rangle$ must be fully alphanumeric, plus underscores and periods

To facilitate our implementation of `validateEmail`, we will define a series of **consumer** functions, which simply serve to consume each of the constituent email parts from the front of a list of characters.

`consumeName : char list -> char list`
REQUIRES: `true`
ENSURES: `consumeName cs` evaluates to the suffix of `cs` caused by removing all characters that could be in a $\langle name \rangle$

`consumeWebsite : char list -> char list`
REQUIRES: `true`
ENSURES: `consumeWebsite cs` evaluates to the suffix of `cs` caused by removing all characters that could be in a $\langle website \rangle$

Consuming Input

```
fun consumeName [] = []
  | consumeName (c::cs) =
      if Char.isAlphaNum c then
        consumeName cs
      else
        case c of
          #"." => consumeName cs
        | #"_" => consumeName cs
        | _ => c::cs
```

```
fun consumeWebsite [] = []
  | consumeWebsite (c::cs) =
      if Char.isAlphaNum c then
        consumeWebsite cs
      else
        c::cs
```

```
fun validateEmail s =
  case consumeName (String.explode s) of
    #"@"::cs =>
      (case consumeWebsite cs of
        #"."::#"o"::#"r"::#"g"::_ => true
      | #"."::#"c"::#"o"::#"m"::_ => true
      | _ => false
      )
  | _ => false
```

Sounds good, right?

No, actually. There's a bug with our code:

```
fun validateEmail s =
  case validateName (String.explode s) of
    #"@"::cs =>
      (case validateWebsite cs of
        [#".", #"o", #"r", #"g"] => true
      | [#".", #"c", #"o", #"m"] => true
      | _ => false
      )
  | _ => false
```

We need to make sure that the extension is the last thing in the email!

How about now?

The above is what we would term a **hand-rolled** function.

This is because it's manually constructed, tedious, and as we will see, more easily solvable in a way that doesn't involve coding a solution from scratch. Hand-rolling is usually undesirable, because more handwritten code means more possibility of errors.

Mantra **More code, more problems.**

This same logic can apply to many situations. For instance, we might be interested in validation for strings which look like home addresses, or strings which look like filenames of images, or strings which are valid social security numbers.

Can we somehow find a way to automate the boilerplate process of writing code like this?

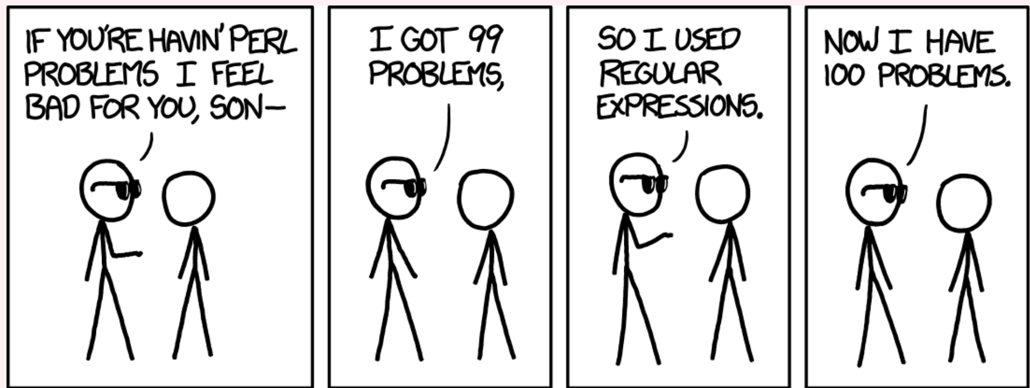
Essentially, we are looking for a family of functions, which would have type `t -> string -> bool`. This type `t` would need to be able to encode the information of the entire form of one of these validation functions, such as `validateEmail`, which could then be given as input to produce the validation function itself.

In essence, we are looking for *the* "essence" of a string validation problem.

This type `t` will end up being the type of **regular expressions**.

2 - Regular Expressions

WHENEVER I LEARN A NEW SKILL I CONCOCT ELABORATE FANTASY SCENARIOS WHERE IT LETS ME SAVE THE DAY.
OH NO! THE KILLER MUST HAVE FOLLOWED HER ON VACATION!
BUT TO FIND THEM WE'D HAVE TO SEARCH THROUGH 200 MB OF EMAILS LOOKING FOR SOMETHING FORMATTED LIKE AN ADDRESS!
IT'S HOPELESS!
EVERYBODY STAND BACK.
I KNOW REGULAR EXPRESSIONS.
PERL!
TAP TAP



[1]

[1] There's an xkcd for everything.

PCRE, which stands for Perl Compatible Regular Expressions, is a particular library and syntax for regular expressions, which allow us to solve certain string matching problems.

We'll take the PCRE, and cut out the PC. We don't have time for that.

At the end of this lecture, you will not know how to actually use regular expressions in the real world, but you will know how they work. That's the important part.

What are the essential elements that make up a "validator"? Recall the email example we were just looking at. There's a few important things that we should be able to do:

- Concatenation - Match one pattern, and then another. For instance, matching the name and then the `@` symbol.
- Alternation - Match possibly one of two patterns. For instance, we could match *either* `org` or `com` at the end.
- Iteration - Match one pattern, as much as you can. For instance, to match a website, we needed to be able to match an alphanumeric character any number of times.

This sure seems like a recursive definition.

Having now described the problem, let's describe the mathematical definition of a regular expression.

Def We say that an **alphabet** is a set of characters, that we are interested in strings composed from. We usually denote this symbolically as Σ.

So for instance, we could have that $\Sigma = \{a, b\}$. Usually, our alphabet of interest will simply be the alphabet of English letters.

Def We denote the set of strings over an alphabet Σ by the symbol Σ^*.

In this case, then we would have that `"abaa"` $\in \Sigma^*$.

Def We say that a **language** L is a subset of Σ^*. In other words, a language is a particular set of strings composed from the alphabet Σ.

So we might say that some examples of languages over $\Sigma = \{a, b\}$ are the empty set, $\{a, aa, aaa, ...\}$, $\{b, bb, bbb, ...\}$, and $\{a, bb\}$.

When we say "string", we mean any finite-length sequence of characters from an alphabet. This includes a string of length 0, which is really hard to write out. As such, as will notate the empty string as ϵ.

Viewed in this notation, what is a validator?

A validator is simply a function which checks for membership within a language. For instance, our `validateEmail` function is simply a function that checks if a string is present in the language of all emails, with emails defined as we said previously.

Def We call a function `f : string -> bool` a **validator** for language L, if `f s` $\Longrightarrow^*$ `true` iff $s \in L$.[2]

We will find that **regular expressions** allow us to implement validators for a class of languages called **regular languages**. These have limitations on their complexity, but in practice a large amount of string validation problems fall into them.

[2] As an aside, this idea of languages and validators over them is very closely tied to the idea of computability! In particular, it is **not** possible, for every language L, to implement a validator for it. A more thorough treatment of this is reserved for a class on computability theory, such as 15-251.

Def Let's first define the structure of a regular expression r:

- 0
- 1
- c, for any character $c \in \Sigma$
- $r_1 + r_2$, for two regular expressions r_1 and r_2
- $r_1 r_2$, for two regular expressions r_1 and r_2
- $r*$, for a regular expression r

If this looks to you like a datatype declaration, that's because soon it will be.

Def We use the notation $L(r)$ to denote the language matched by regular expression r, over some alphabet Σ. Then:

Construct	Language matched
$L(c)$	$\{c\}$
$L(0)$	$\{\}$
$L(1)$	$\{\epsilon\}$
$L(r_1 + r_2)$	$L(r_1) \cup L(r_2)$
$L(r_1 r_2)$	$\{s_1 s_2 \mid s_1 \in L(r_1),\ s_2 \in L(r_2)\}$
$L(r^*)$	$\{s_1 ... s_n \mid \text{for } n \geq 0, \text{when } \forall i, s_i \in L(r)\}$

The Language of a Regular Expression, In English

Construct	Matches
$L(c)$	only c
$L(0)$	nothing
$L(1)$	only the empty string
$L(r_1 + r_2)$	anything matched by either r_1 or r_2
$L(r_1 r_2)$	anything with a prefix matched by r_1 and suffix matched by r_2
$L(r^*)$	any string which is something matched by r, 0 or more times

Let's look at some examples of regular expressions and the languages they match.

In this example, let's assume we are working with the alphabet $\Sigma = \{a, b\}$.

Regular Expression	Language matched
$a + b$	$\{a, b\}$
$abaa + baa$	$\{abaa, baa\}$
$(abaa + baa)b$	$\{abaab, baab\}$
$0 + a$	$\{a\}$
$1 + a$	$\{\epsilon, a\}$
a^*	$\{\epsilon, a, aa, aaa, aaaa, \ldots\}$

Let's ground ourselves a little bit. Why are we talking about this?

Regular expressions are a useful formalism that allow us to symbolically specify certain languages. They are composable, since we can easily form regular expressions out of other ones. We haven't yet specified how to turn this formalism into code that runs, but this is the model by which we will design our code.

In particular, recall the website example that we were talking about previously. We can express our website validator by the following regular expression:

$$(r_{an} + . + _)^{*}@(r_{an}{}^{*}).(org + com)$$[3]

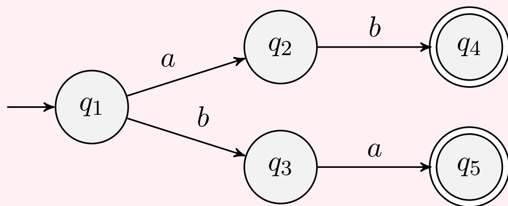
where r_{an} is the regular expression of all alphanumeric characters, which could be specified as $a + b + ... + z + 1 + 2 + ... + 9 + 0$.

[3] This is actually not quite it, because this regular expression allows an empty name or website. But for the sake of brevity, we'll go with this.

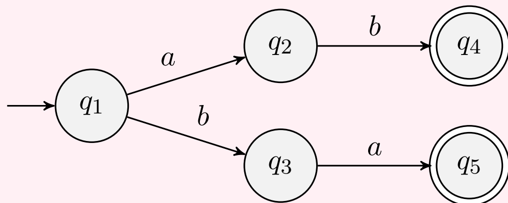
3 - Finite-State Automata

It is a very interesting fact that regular expressions can be viewed as **finite-state machines**, which are graph-like constructs with **states**, which change upon receiving characters as input. In particular, we can characterize them by **deterministic finite-state automata**.[4]

For instance, here is a finite-state machine corresponding to the regex $ab + ba$:



[4] We can characterize them by **nondeterministic** finite-state automata as well. But this, too, will be left in more detail to a class on computability.

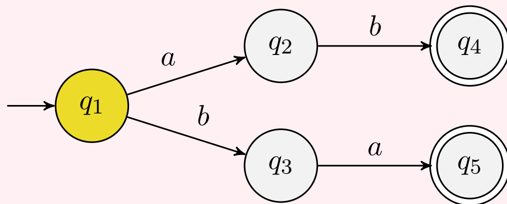


Here, we denote the **starting state** by the state with an unlabeled in-arrow.

From a given state, transitions to other states upon reading a particular character are denoted by arrows labeled with characters. If there is no arrow corresponding to the input character, then the entire string is rejected.

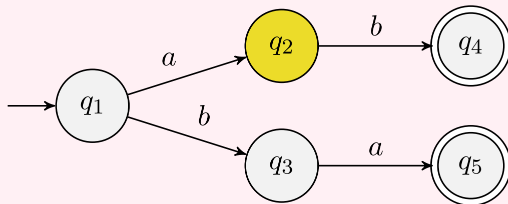
We also say that a state is an **accepting state** if it is a node which has a circle in it. If the state reached upon reading the entire input is an accepting state, then we say that that string is in accepted by the FSM.

Let's try giving this DFA an input, namely the string ab.



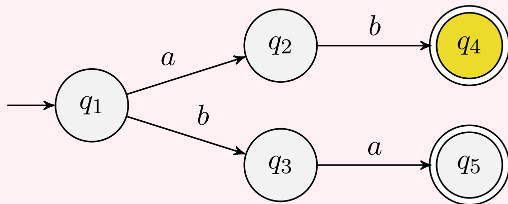
Remaining input: ab

After reading a:



Remaining input: b

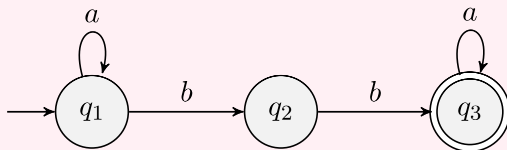
After reading b:



No remaining input, so we end up in state q_4, which is accepting.

So the string ab is accepted by this DFA.

Here's another DFA for the regular expression a^*bba^*:



You should be able to convince yourself that this automaton accepts all strings which contain any number of as enclosing two bs.

4 - Matching Regular Expressions

We've seen that, using the DFA for a regular expression, we can visualize a computational process for validating strings within a particular language.

This process is an extremely beautiful intersection where theory and practice coincide, and is how many production regular expression engines[5] work today. However, the process of producing such an automaton from a regular expression is rather involved. We will take another track for how to produce a validator from a regular expression.

[5] https://github.com/google/re2

We notice that regular expressions are a recursive datatype – that is, regular expressions are composed out of regular expressions.

We can define the type of regular expressions as follows:

```
datatype regexp =
    Zero
  | One
  | Char of char
  | Plus of regexp * regexp
  | Times of regexp * regexp
  | Star of regexp
```

The definition of the language of a regular expression is a straightforward recursive definition, which depends on the language of the regular sub-expressions. Let's try to write a function which can recursively decompose on a regular expression, and in a backtracking way, try to match a string.

We will find that this is a good application of CPS. We will implement a function:

```
match : regexp -> char list -> (char list -> bool) -> bool
```

REQUIRES: `k` is total

ENSURES:

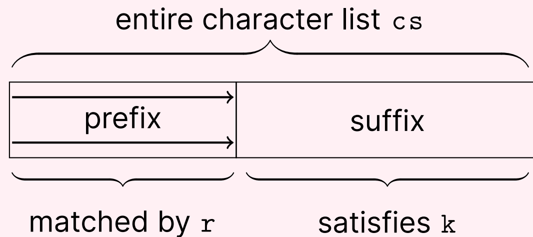
$$\texttt{match r s k} \cong \begin{cases} \texttt{true}, & \text{if } \texttt{cs} \cong \texttt{p @ s} \text{ where } \texttt{p} \in L(r) \text{ and } \texttt{k s} \cong \texttt{true} \\ \texttt{false}, & \text{otherwise} \end{cases}$$

This specification is kind of complicated. Can we desugar it at all?

We will find that this type makes our implementation a lot simpler. We use a continuation, specific to return type `bool`, to denote a *future condition* we are placing upon the rest of the character list. This is useful for branching possibilities, because it lets us enforce a future condition on potentially many suffixes that we might choose to consume input to obtain.

For instance, we might be interested in something which looks like `match r cs (fn cs' => List.length cs' = 3).`

This essentially is a nondeterministic search over *all possible prefixes* that can be taken by `match`, by the regex `r`, *provided* that the suffix to that prefix is of length 3. We change the continuation in order to enforce a condition on all of the possibilities we might pick.



Note that the prefix goes from left to right, as we gradually take off more and more of the character list, trying out prefixes to see if they will eventually work.

```
fun match (r : regexp) (cs : char list) (k : char list ->
   bool) : bool =
  case r of
    Zero => (* ... *)
  | One => (* ... *)
  | Char c => (* ... *)
  | Plus (r1,r2) => (* ... *)
  | Times (r1, r2) => (* ... *)
  | Star r => (*... *)
```

Let's write the cases for `Zero` and `One` first.

We know that `Zero` matches the empty language, so there is no possible prefix that we can take. So we must reject.

```
Zero => false
```

For `One`, we only allow the empty string. So our prefix must necessarily be empty, meaning that the only way to return true is if the entire list satisfies the continuation.

```
| One => k cs
```

For the `Char` case, we can actually start to take inputs off from the list. We know that for the regexp `Char c`, the only string in that language is `c`, so the only prefix we can take is the singleton list `c`.

So we write:

```
| Char c => (case cs of
    [] => false
  | c' :: cs' => c = c' andalso k cs')
```

because in the empty case, there is no such prefix, and in the cons case, we still need to make sure the suffix satisfies the continuation.

What about the `Plus (r1, r2)` case? Here, we have the possibility of picking a prefix from either `r1` or `r2`.

Thankfully, the return type of our function `match` is just `bool`, and we're not required to write `match` tail recursively, so we can simply use two recursive calls:

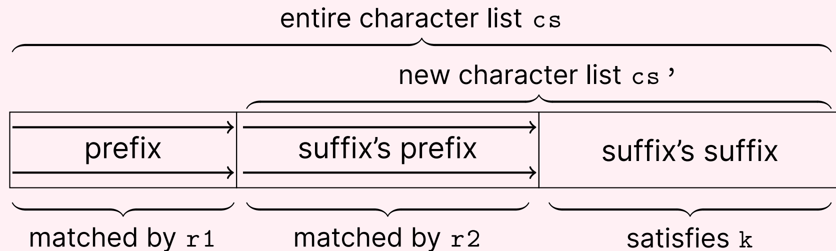
```
| Plus (r1,r2) => match r1 cs k orelse match r2 cs k
```

We are essentially searching over all the strings matched by `r1` and `r2` separately, but with the same suffix condition needing to be true. Either succeeding means that we succeed in general.

For concatenation, we need to be able to pick a prefix from the `cs` for the first regexp `r1`, but then pick another prefix after that for `r2`.

This is exactly the same as looking for a prefix of `cs`, but then looking for a prefix of the corresponding suffix. Let's write it:

```
| Times (r1, r2) => match r1 cs (fn cs' => match r2 cs' k)
```



For the `Star r` case, we need to somehow be able to take 0 or more prefixes, all of which match the `r` regexp.

This means that when we make a recursive call to `match` to find a single prefix, our continuation on the corresponding suffix needs to be able to furthermore take more prefixes of what's left. In fact, we might not even need to find a single prefix.

We make an observation: the regular expression r^* is the same as either matching the empty string, or matching r one or more times.

Essentially, we are saying that $L(r^*) = L(1 + rr^*)$.

```
| Star r =>
    k cs orelse match r cs (fn cs' => match (Star r) cs' k)
```

However, there's something fishy here. Can you spy it?

```
| Star r =>
    k cs orelse match r cs (fn cs' => match (Star r) cs' k)
```

In the continuation, we call `match` again on precisely the same arguments, except for `cs'`. Is it possible, however, that `cs'` might be the same as `cs`?

Answer: Yes, because our prefix might be empty!

In general, you should be suspicious whenever you see a recursive call which might have arguments which do not change. That's a surefire way to an infinite loop!

We see that the case of `r`, where $\epsilon \in L(\texttt{r})$, can cause a problem with this implementation. For example, the input `match (Star One) [#"a"] List.null` will loop forever. How can we solve this problem?

There are two ways. We can either **weaken the specification** or **strengthen the implementation**.

What does *weakening the specification* mean? It means that, instead of claiming an ambitious postcondition or a minimal precondition, we can either claim to do less, by promising less in our postcondition, or add more caveats, by adding more restrictions to our precondition.

What does *strengthening the implementation* mean? It means taking extra care by writing more code so that we can fulfill our postconditions and preconditions as-is, without needing to make any compromises.

For instance, suppose we have the `fact` function.

```
fun fact 0 = 1
  | fact n = n * fact (n - 1)
```

We loop forever on negative inputs, which is obviously undesirable. To solve this, we can:

- *strengthen the implementation* by making it return an `option` on a negative input, or
- *weaken the specification* by merely adding a precondition that negative inputs are not permitted.

Both are valid ways, depending on how the function is used.

So how could we weaken the specification for our regular expression matcher? We could require that the regular expression be passed in in a form such that there are no occurrences of `Star r`, where $\epsilon \in L(\mathtt{r})$. This would be a cheap way of escaping, by pushing the burden onto the caller.[6]

Another way is that we can strengthen the implementation, by making our `match` function able to deal with the case where our prefix is the empty string. We can do that by simply observing that the suffix is the same as the original `char list`.

We will take this approach for now.

[6] It turns out this can be done. We say that such regular expressions are in **standard form**. There is a programmatic way of turning any regular expression into one in standard form, but which matches the same language.

```
| Star r =>
    k cs orelse match r cs (fn cs' => cs' <> cs andalso match
    (Star r) cs' k)
```

If not previously mentioned before, the `<>` is the polymorphic inequality operator.

Now, we check that our new suffix `cs'` is not the same as the entire `char list`! This way, we ensure that we always make progress, and because of short-circuiting `andalso`, we ensure we never proceed to the recursive call on `match`.

```
fun match (r : regexp) (cs : char list) (k : char list -> bool) :
   bool =
  case r of
    Zero => false
  | One => k cs
  | Char c => (case cs of
      [] => false
    | c' :: cs' => c = c' andalso k cs')
  | Plus (r1,r2) => match r1 cs k orelse match r2 cs k
  | Times (r1, r2) => match r1 cs (fn cs' => match r2 cs' k)
  | Star r =>
      k cs orelse match r cs (fn cs' => cs' <> cs andalso match (
   Star r) cs' k)
```

Now, we can define our function `accept`, using `match`:

accept : regexp -> string -> bool
REQUIRES: `true`
ENSURES: `accept r s` $\cong$ `true` iff `s` $\in L(\texttt{r})$, and `false` otherwise.

This comes from a simple observation that to accept a string, we have to make sure that we match the entire string. In other words, the suffix is empty:

```
fun accept r s = match r (String.explode s) List.null
```

Super concise.

5 - Proving Correctness

Recall the specification of `match`.

```
match : regexp -> char list -> (char list -> bool) -> bool
```

REQUIRES: `k` is total

ENSURES:

$$\texttt{match r s k} \cong \begin{cases} \texttt{true}, & \text{if } \texttt{cs} \cong \texttt{p @ s} \text{ where } \texttt{p} \in L(r) \text{ and } \texttt{k s} \cong \texttt{true} \\ \texttt{false}, & \text{otherwise} \end{cases}$$

This is a hefty one, but can we prove that it's correct?

Proving the Specification

Because `match` returns a `bool` at the end of everything, there are four possible behaviors that we are concerned with.

It can either:

- return the value `true`
- return the value `false`
- loops forever
- raises an exception

We would like to show that it only performs the first two behaviors, and only in the right circumstances.

It is surprisingly hard to prove that `match` is total. We will assume that we have already done so, because the proof is long and involved.

Lemma `match r cs k`, for total `k`, is always valuable.

Assuming that `match r cs k` always terminates, we only have the following two behaviors:

- return the value `true`
- return the value `false`

In this world, then we only need to show the following theorem:

Thm. For total `k`, `match r cs k`, returns `true` iff `cs` $\cong$ `p @ s` where `p` $\in L(r)$, and `k s` $\cong$ `true`

We call the forward implication **soundness**, and the reverse implication **completeness**.

Recall that we prove a bi-implication by proving each of the implications separately. Thus, proving our theorem reduces to proving soundness and completeness separately.

We will now prove this theorem by structural induction on `r : regexp`.

We have three base cases: `Zero`, `One`, and `Char c`.

We also have three inductive cases: `Plus (r1, r2)`, `Times (r1, r2)`, and `Star r`. We will have inductive hypotheses tailored to the sub-regexes in each case, for each branch of the proof.

For now, we will only prove the `Plus` case.

In the following two proofs, Let $P(\mathtt{r})$ be the following:

Thm. For total `k`, `match r cs k` $\hookrightarrow$ `true` if and only if `cs` $\cong$ `p @ s` where `p` $\in L(r)$, and `k s` $\cong$ `true`.

We would like to show the forward direction of P(`Plus (r1, r2)`), which is:

If `match (Plus (r1, r2)) cs k` $\hookrightarrow$ `true`, then
`cs` $\cong$ `p @ s` where `p` $\in L$(`Plus (r1, r2)`), and `k s` $\cong$ `true`.

Let us assume that `match (Plus (r1, r2) cs k)` $\cong$ `true`.
Assume for our induction hypotheses the forward directions of P(`r1`) and P(`r2`).
In particular, P(`r1`) reads:

If `match r1 cs k` $\cong$ `true`, then
`cs` $\cong$ `p @ s` where `p` $\in L$(`r1`), and `k s` $\cong$ `true`.

Then:

`true` $\cong$ `match (Plus (r1, r2)) cs k` (our assumption)

$\cong$ `match r1 cs k orelse match r2 cs k` (def of `match`)

By the specification of `orelse`, this must mean that either `match r1 cs k` or `match r2 cs k` $\cong$ `true`.

Without loss of generality[7], assume that `match r1 cs k` $\cong$ `true`. Then, we can apply our induction hypothesis $P(\texttt{r1})$, and obtain that

`cs` $\cong$ `p @ s` where `p` $\in L(\texttt{r1})$ and `k s` $\cong$ `true`.

By the definition of $L(\texttt{Plus(r1, r2)})$, this means that `p` $\in L(\texttt{Plus(r1, r2)})$.

With both the previous and our inductive conclusion, then this is exactly what we wanted to show, so we have proven the theorem.

[7] This is a fancy CMU way of saying that the proof is similar in either case, so we're only going to bother to prove it for one.

We would like to show the backwards direction of $P(\texttt{Plus (r1, r2)})$, which is:

If $\texttt{cs} \cong \texttt{p @ s}$ where $\texttt{p} \in L(\texttt{Plus(r1, r2)})$, and $\texttt{k s} \cong \texttt{true}$, then
$\texttt{match (Plus (r1, r2)) cs k} \cong \texttt{true}$

Let us assume that
$\texttt{cs} \cong \texttt{p @ s}$ where $\texttt{p} \in L(\texttt{Plus (r1, r2)})$, and $\texttt{k s} \cong \texttt{true}$.

Assume for our induction hypotheses the reverse directions of $P(\texttt{r1})$ and $P(\texttt{r2})$.
In particular, $P(\texttt{r1})$ reads:

If $\texttt{cs} \cong \texttt{p @ s}$ where $\texttt{p} \in L(\texttt{r1})$, and $\texttt{k s} \cong \texttt{true}$, then
$\texttt{match r1 cs k} \hookrightarrow \texttt{true}$.

Completeness Proof

By the definition of $L(\texttt{Plus (r1, r2)})$, this assumption means that either $\texttt{p} \in L(\texttt{r1})$ or $\texttt{p} \in L(\texttt{r2})$. Without loss of generality, assume that $\texttt{p} \in L(\texttt{r1})$.

Then, we can apply our induction hypothesis $P(\texttt{r1})$, since we know that $\texttt{cs} \cong \texttt{p @ s}$ where $\texttt{p} \in L(\texttt{r1})$, and $\texttt{k s} \cong \texttt{true}$. So $\texttt{match r1 cs k} \hookrightarrow \texttt{true}$.

Then:

$$
\begin{aligned}
&\texttt{match (Plus (r1, r2)) cs k} \\
&\cong \texttt{match r1 cs k orelse match r2 cs k} && \text{(def of match)} \\
&\cong \texttt{true} && \text{(our previous conclusion)}
\end{aligned}
$$

So we have proven the theorem.

Thank you!