Regular Expressions

Language Hierarchy

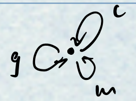
Unrestricted

Context - sensitive

Context - free

Regular { recognizer: finite Automata
Application: Tokenization

e.g. excursion from home



c: go to CMU, then go home
g: go to grocery, then go home
w: go for a walk, then go home

c: go to CMU once
cc: go to CMU twice
...

c^*: ≥ 0 times go to CMU

g+w: get groceries OR a walk

$(g+w)^*$: ≥ 0 times get groceries OR a walk

Σ : an alphabet of characters

Σ^* : the set of all finite-length strings over Σ (i.e. with chars in Σ)

e.g. "aabba" is in $\{a, b\}^*$

ε : the empty string, containing no chars

A language over Σ is a subset of Σ^* (may contain infinite strings, but via a finite representation)
regular expressions

A regular expression over Σ is any of the following:

a for every $a \in \Sigma$

0 a special symbol

1 a special symbol

$r_1 + r_2$ with r_1, r_2 regular expressions alternation

$r_1 r_2$ with r_1, r_2 regular expressions concatenation

r^* with r a regular expression kleene star

Regular Language

Given regular expression r, language: $L(r)$

$L(a) = \{a\}$ singleton set $\forall a \in \Sigma$

$L(0) = \{\ \}$ empty language, no strings

$L(1) = \{\varepsilon\}$ consist of empty string

$L(r_1 + r_2) = \{ s \mid s \in L(r_1) \text{ or } s \in L(r_2) \}$

$L(r_1 r_2) = \{ s_1 s_2 \mid s_1 \in L(r_1) \text{ and } s_2 \in L(r_2) \}$

$L(r^*) = \{ s \mid s = s_1 s_2 \ldots s_n,\ n \geq 0 \text{ with each } s_i \in L(r) \}$ $s = \varepsilon$ when $n = 0 \Rightarrow \varepsilon \in L(r^*)\ \forall r$

Let L be a subset of Σ^*. L is regular if $L = L(r)$ for some regular expression r.
minimal class closed under

Assume $\Sigma = \{a, b\}$

$L(a) = \{a\}$ $L((a+b)^*) = \Sigma^*$

$L(aa) = \{aa\}$ $L((a+b)^* aa (a+b)^*) =$ all strings in Σ^* containing ≥ 2 consecutive as.

$L((a+1)(b+ba)^*) =$ all strings in Σ^* not containing ≥ 2 consecutive a

} negation !

$L(\underbrace{r^*}_{\text{0 or more}}) = L(\underbrace{1}_{0}+\underbrace{rr^*}_{\text{1 or more}})$

Acceptor

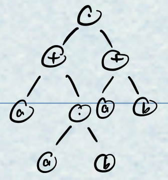
```
(* accept : regexp -> string -> bool
   REQ: true (?)
   ENS: (accept r s) => true if s ∈ L(r)
        (accept r s) => false otherwise
*)

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

$\underbrace{\texttt{String.explode}}_{\text{turn string into char list}}$ $\underbrace{\texttt{List.null}}_{\text{'a list -> bool}}$

Suppose $r = (a+ab)(a+b)$

$L(r) = \{aa, ab, aba, abb\}$



Split aba :

(_, aba), (a, ba), (ab, a)
× × ✓

Matcher

```
(* match : regexp -> char list -> (char list -> bool) -> bool
   REQ : k is total (?)
   ENS : match r cs k => true, if cs ≅ p@s with p ∈ L(r) and k(s) ↪ true
                      => false, otherwise
*)
```

$\underbrace{\texttt{char list -> (char list -> bool) -> bool}}_{\text{matcher}}$

```
datatype regexp =

fun match (Char a) cs k = (case cs of
                             [] => false      a ∉ []
                           | c::cs => (a = c) andalso (k cs'))
  | match Zero _ _ = false
  | match One cs k = k cs
  | match (Plus (r1,r2)) cs k = (match r1 cs k) orelse (match r2 cs k)
  | match (Times (r1,r2)) cs k = (match r1 cs k) (fn cs' => match r2 cs' k)
  | match (Star r) cs k = k cs orelse match r cs (fn cs' => match (Star r) cs' k))   ?
                                                          ^
                                    not (cs = cs') andalso / assume std form in REQ
```

$L(r^*) = L(1+rr^*)$

match (Star One) [#"a"] List.null -> loop forever

since List.null [#"a"] ≅ false, match One cs k' passes all cs' to match

How to fix ? cs' must be a proper suffix of cs! Change spec / code!

Proof

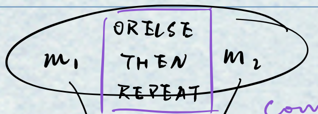
1. prove termination (assume true)
2. prove soundness & completeness   if and only if

```
match r cs k => true, if cs ≅ p@s with p ∈ L(r) and k(s) ↪ true
             => false, otherwise
```

Base Case: Zero, One, Char(a) ∀ a : char

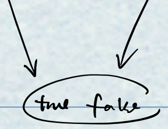
IS: Plus, Times, Star

eg. Plus



combinators

type matcher = char list -> (char list -> bool) -> bool



orelse

andalso

Base Case REJECT ACCEPT CHECK_FOR Combinators ---

match

```
                  (char list -> bool) -> bool

val REJECT : matcher = fn _ => fn _ => false   always reject
val ACCEPT : matcher = fn cs => fn k => k cs   not always accept (locally accept, pass into continuation
                                                                   to check the real outcome)

fun CHECK_FOR (a : char) : matcher cs k = (case cs of
    (Version 1)                                      [] => false
                                                   | c::cs' => a = c andalso k cs')

fun CHECK_FOR (a : char) : matcher =
    (Version 2) pass the work into continuation

    fn [] => REJECT []                                        the input doesn't matter
     | c::cs' => if a = c then ACCEPT cs' otherwize REJECT (c::cs')
char list -> (char list -> bool) -> bool
```

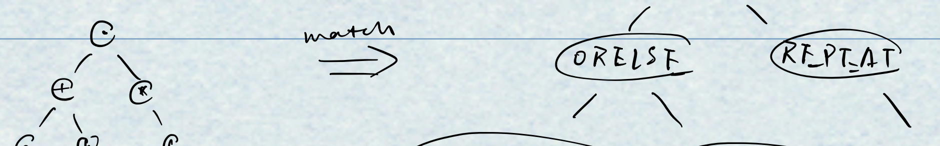
```
infix 8 ORELSE                      ORELSE : matcher * matcher -> matcher
infix 9 THEN                        THEN   : matcher * matcher -> matcher
                                    REPEAT : matcher -> matcher
fun (m1 OREISE m2) cs k = m1 cs k orelse m2 cs k
fun (m1 THEN m2) cs k = m1 cs (fn cs' => m2 cs' k)
fun REPEAT m1 cs k = (k cs) orelse m (fn cs' => REPEAT m cs' k)
```

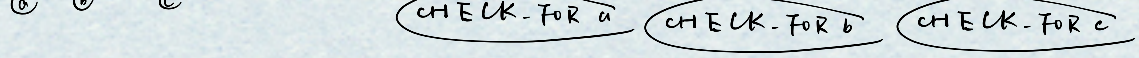
```
(Star Version 2)
| match (Star r) cs k = let
                          fun mstar cs' = (k cs') orelse (match r cs' mstar)
                        in
                          mstar cs
                        end
```

```
(Similarly)
fun REPEAT m cs k =   let
                        fun mstar cs' = (k cs') orelse (m cs' mstar)
                      in
                        mstar cs
                      end
```

```
(Rewrite match)
fun match (Char a : regexp) : matcher = CHECK_FOR a
  | match Zero _ _ = REJECT
  | match One cs k = ACCEPT                                       seperate regular exp
  | match (Plus (r1, r2)) cs k = match r1 ORELSE match r2         from other implementations
  | match (Times (r1, r2)) cs k = match r1 THEN match r2
  | match (Star r) cs k = REPEAT match r
```

$r = (a+b)c^*$





```
(* accept: regexp -> string -> bool *)
fun accept (r : regexp) : string -> bool =  fn s => match r (String.explode s) List.null
```

⇓ staging

```
fun accept (r : regexp) : string -> bool = let
                                             val m = match r
                                           in
                                             fn s => m (String.explode s) List.null
                                           end
```