15-150
Fall 2024

Dilsun Kaynar

LECTURE 15

Regular Expressions
(using combinators as staging)

Representing regular expressions

$$a \mid 0 \mid 1 \mid r_1\, r_2 \mid r_1 + r_2 \mid r^*$$

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

Review

accept and match

```
(* accept : regexp -> string -> bool

   REQUIRES:  true
   ENSURES:   (accept r s)  ≅ true, if s ∈ L(r);
              (accept r s)  ≅ false, otherwise.
*)


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

   REQUIRES: k is total.
   ENSURES:  (match r cs k)  ≅ true,
                        if cs can be split as cs ≅ p@s,
                        with p representing a string in L(r)
                        and k(s) evaluating to true;
             (match r cs k)  ≅ false, otherwise.
*)

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

$L(a) = \{a\}$
$L(0) = \{\}$
$L(1) = \{\varepsilon\}$
$L(r_1\ r_2) = \{s_1\ s_2 \mid s_1 \in L(r_1) \text{ and } s_2 \in L(r_2)\}$
$L(r_1 + r_2) = \{s \mid s \in L(r_1) \text{ or } s \in L(r_2)\}$
$L(r^*) = \{s_1 \ldots s_n \mid n \geq 0 \text{ with } s_i \in L(r) \text{ for } 0 \leq i \leq n\}$
Alternatively,
$L(r^*) = \{\varepsilon\} \cup \{s_1 s_2 \mid s_1 \in L(r) \text{ and } s_2 \in L(r^*)\}$

```
fun match (Char(a)) cs k =  (case cs of
                                [ ] => false
                              | (c::cs') => (a=c) andalso k(cs'))
  | match (Zero) _ _ =   false
  | match (One) cs k =  k(cs)
  | match (Times (r1,r2)) cs k =  match r1 cs (fn cs' => match r2 cs' k)
  | match (Plus (r1,r2)) cs k =   match r1 cs k orelse match r2 cs k
  | match (Star(r)) cs k =   k(cs) orelse match r cs (fn cs' => match Star(r) cs' k)
```

may lead to an infinite loop

Example: match(Star(One)) ["#a"] List.null

List.null ["#a"] is false and match One cs k' will pass cs to k'

Two ways to fix the problem

- Change code
- Change specification to require that the input regular expression be in *standard form*
  - If Star(r) appears in the regular expression then ε is not in the language of r.

```
fun match (Char(a)) cs k =  (case cs of
                                [ ] => false
                              | (c::cs') => (a=c) andalso k(cs'))
  | match (Zero) _ _ =  false
  | match (One) cs k =  k(cs)
  | match (Times (r1,r2)) cs k =  match r1 cs (fn cs' => match r2 cs' k)
  | match (Plus (r1,r2)) cs k = match r1 cs k orelse match r2 cs k
  | match (Star (r)) cs k = k(cs) orelse match r cs
                                   (fn cs’ => not (cs = cs’)
                                              andalso match Star(r) cs’ k)
```

Or we could check cs’ gets smaller

```
fun match (Char(a)) cs k =   (case cs of
                                  [ ] => false
                                | (c::cs') => (a=c) andalso k(cs'))
  | match (Zero) _ _ =   false
  | match (One) cs k =   k(cs)
  | match (Times (r1,r2)) cs k =   match r1 cs (fn cs' => match r2 cs' k)
  | match (Plus (r1,r2)) cs k = match r1 cs k orelse match r2 cs k
  | match (Star (r)) cs k =   k(cs) orelse match r cs  (fn cs' => match Star(r) cs' k)
```

Or we could require that r be in standard form

A regular expression r is in *standard form* if and only if for any subexpression Star(r') of r, L(r') does not contain the empty string.

L(r*) = L(1 + r r*)

Sketch of a Proof of Correctness

- **Prove termination:** show that (match r cs k) returns a value for all arguments r, cs, k satisfying REQUIRES (We will assume this).

- **Prove soundness and completeness:** (We will do this assuming termination and write out one case).

Soundness and Completenes (assuming termination)

ENSURES: (match r cs k) ≅ true, if cs ≅ p@s,
with p ∈ L(r) and k(s) ≅ true;
(match r cs k) ≅ false, otherwise

Given termination, we can rephrase the spec as follows:

ENSURES: (match r cs k) ≅ true if and only if there exist p, s such that
cs ≅ p@s, p ∈ L(r) and k(s) ≅ true

Theorem:

For all values r: regexp, cs: char list, k: char list -> bool, with k total

(match r cs k) ≅ true

if and only if

there exist p, s such that

cs ≅ p@s, p ∈ L(r) and k(s) ≅ true

We are assuming termination as a lemma.

Proof: By structural induction on r

Base cases: Zero, One, Char (a) for every a: char

Inductive cases: Plus (r_1, r_2), Times (r_1,r_2), Star (r)

Theorem:

For all values r: regexp, cs: char list, k: char list -> bool, with k total

(match r cs k) ≅ true

if and only if

there exist p, s such that

cs ≅ p@s, p ∈ L(r) and k(s) ≅ true

We are assuming termination as a lemma.

Inductive case: r = Plus (r_1, r_2) for some r_1 and r_2

IH: For i = 1,2, for all values cs: char list, k: char list -> bool, with k total, (match r_i cs k) ≅ true if and only if there exist p, s such that cs ≅ p@s, p ∈ L(r_i) and k(s) ≅ true

NTS: For all values cs: char list, k: char list -> bool, with k total, (match (Plus (r_1, r_2)) cs k) ≅ true if and only if there exist p, s such that cs ≅ p@s, p ∈ L(Plus (r_1, r_2)) and k(s) ≅ true.

Soundness

Inductive case: $r = \text{Plus}\ (r_1, r_2)$ for some r_1 and r_2

IH: For i = 1,2, for all values cs: char list, k: char list -> bool, with k total (match r_i cs k) ≅ true if and only if there exist p, s such that cs ≅ p@s, p ∈ L(r_i) and k(s) ≅ true

NTS: For all values cs: char list, k: char list -> bool, with k total (match (Plus (r_1, r_2)) cs k)) ≅ true if and only if there exist p, s such that cs ≅ p@s, p ∈ L(Plus (r_1, r_2)) and k(s) ≅ true.

(Part 1): Suppose (match (Plus (r_1, r_2)) cs k) ≅ true

NTS: There exist p, s such that
such that cs ≅ p@s, p ∈ L(Plus (r_1, r_2)) and k(s) ≅ true.

true ≅ (match (Plus (r_1, r_2)) cs k) [Assumption]

≅ (match r_1 cs k) **orelse** (match r_2 cs k) [Plus]

One or both arguments to **orelse** must be true. Let's suppose the first one.

By IH for r_1 there exist p, s such that cs ≅ p@s, p ∈ L(r_1) and k(s) ≅ true.

p ∈ L(Plus (r_1, r_2)) by language definition for Plus.

Completeness

Inductive case: r = Plus (r_1, r_2) for some r_1 and r_2

IH: For i = 1,2, for all values cs: char list, k: char list -> bool, with k total (match r_i cs k) $\cong$ true if and only if there exist p, s such that cs $\cong$ p@s, p $\in$ L(r_i) and k(s) $\cong$ true

NTS: For all values cs: char list, k: char list -> bool, with k total (match (Plus (r_1, r_2)) cs k) $\cong$ true if and only if there exist p, s such that cs $\cong$ p@s, p $\in$ L(Plus (r_1, r_2)) and k(s) $\cong$ true.

(Part 2): Suppose cs $\cong$ p@s, p $\in$ L(Plus (r_1, r_2)) and k(s) $\cong$ true.

NTS: (match (Plus (r_1, r_2)) cs k) $\cong$ true

(match (Plus (r_1, r_2)) cs k)

$\cong$ (match r_1 cs k) **orelse** (match r_2 cs k) [Plus]

By supposition, there exist p, s such that cs $\cong$ p@s, p $\in$ L(Plus (r_1, r_2)) and k(s) $\cong$ true. By language definition for Plus, p $\in$ L(r_1) and/or p $\in$ L(r_2).
If p $\in$ L(r_1), then (match r_1 cs k) $\cong$ true, by IH for r_1.
Otherwise, p $\in$ L(r_2), (match r_1 cs k) $\cong$ false by termination,
and (match r_2 cs k) $\cong$ true by IH for r_2.

Using combinators

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

Space of functions that return booleans

Space of booleans

Idea: interpret the syntax of regular expressions as operations on matchers.

Code design

- match will take a regular expression and return a function (matcher) of type char list -> (char list -> bool) -> bool

- Combine functions of this type using combinators

    - Stage 1: Deconstructing regular expressions by pattern matching

    - Stage 2: Deal with the input string

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

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

Recall the staging example

```
fun f (x:int) : int -> int =
    let
        val z: int = horrible(x)
    in
        fn y => z + y
    end
```

value of horrible(x) is bound to z in the environment of the returned function

Recall the staging example

```
fun accept (r) =
      let
          val m = match (r)
      in
          fn s: string => m ....
     end
```

Build a matcher from a regexp

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

Using a combinator library with functions of this type

```
fun match (Char a) =  CHECK_FOR a
   | match Zero =  REJECT
   | match One =  ACCEPT
   | match (Times (r1, r2)) =(match r1) THEN (match r2)
   | match (Plus (r1, r2)) = (match r1) ORELSE (match r2)
   | match (Star r) =  REPEAT (match r)
```

One can produce a matcher for a regular expression without ever seeing any input or continuations

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

Continuation base cases

instantly fail

```
val REJECT : matcher =  fn cs => fn k => false

val ACCEPT : matcher = fn cs => fn k => k (cs)
```

call the continuation

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

Continuation base cases

```
val REJECT : matcher =  fn cs => fn k => false

val ACCEPT : matcher = fn cs => fn k => k (cs)
```

Suppose we had written REJECT without type annotations. What would its type be?

'a -> 'b -> bool

Suppose we had written ACCEPT without type annotations. What would its type be?

'a -> ('a -> 'b) -> 'b

Build a matcher from a regexp

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

Using a combinator library with functions of this type

```
fun match (Char a) = CHECK_FOR a
  | match Zero = REJECT
  | match One = ACCEPT
  | match (Times (r1, r2)) = (match r1) THEN (match r2)
  | match (Plus (r1, r2)) = (match r1) ORELSE (match r2)
  | match (Star r) = REPEAT (match r)
```

Input related

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

```
val REJECT : matcher = fn cs => fn k => false

val ACCEPT : matcher = fn cs => fn k => k (cs)
```

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

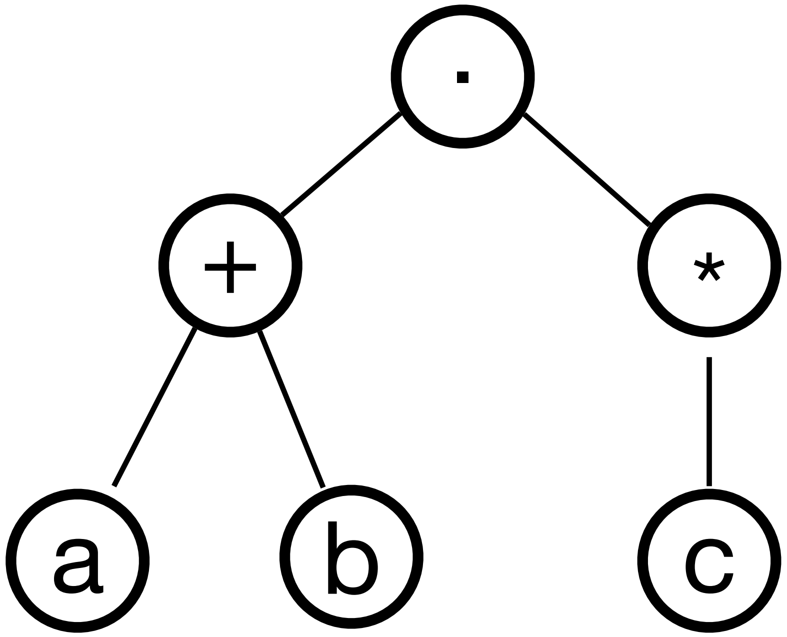
```
(* Alternatively, using REJECT and ACCEPT *)

fun CHECK_FOR (a : char) : matcher =
     fn [ ] =>________
      | c::cs => if a=c then _________
                 else _________
```

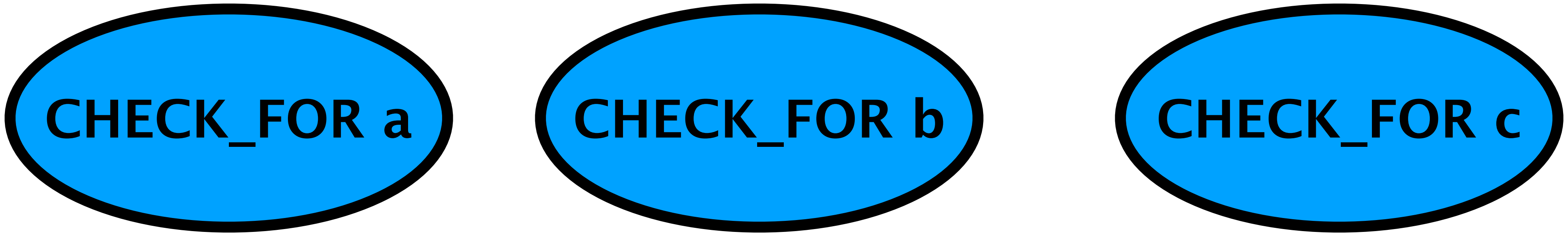
```
val REJECT : matcher = fn cs => fn k => false

val ACCEPT : matcher = fn cs => fn k => k (cs)
```

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

```
(* Alternatively, using REJECT and ACCEPT *)

fun CHECK_FOR (a : char) : matcher =
    fn [ ] => REJECT [ ]
     | c::cs => if a=c then ACCEPT cs
                else REJECT (c::cs)
```

(a+b) c*





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

ORELSE and THEN

```
infixr 8 ORELSE
infixr 9 THEN

fun (m1 : matcher) ORELSE (m2 : matcher) : matcher =
        fn cs => fn k => m1 cs k orelse m2 cs k

fun (m1 : matcher) THEN (m2 : matcher) : matcher =
        fn cs => fn k => m1 cs (fn cs' => m2 cs' k)
```

Recall the match (Star (r))

```
fun match (Char(a)) cs k =  (case cs of

  | ............................

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



  (* Alternatively, ... *)

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

It avoids packing and unpacking r with Star

REPEAT

Assuming that regular expressions are in standard form

```
fun REPEAT (m : matcher) : matcher = fn cs => fn k =>
  let
    fun mstar cs' = ____________________
  in
    mstar cs
  end
```

```
fun match (Char a) = CHECK_FOR a
  | match Zero = REJECT
  | match One = ACCEPT
  | match (Times (r1, r2)) =(match r1) THEN (match r2)
  | match (Plus (r1, r2)) = (match r1) ORELSE (match r2)
  | match (Star r) = REPEAT (match r)
```

REPEAT

Assuming that regular expressions are in standard form

```
fun REPEAT (m : matcher) : matcher = fn cs => fn k =>
  let
    fun mstar cs' = k cs' orelse m cs' mstar
  in
    mstar cs
  end
```

Exercise

Write evaluation steps for accept (Plus(Char(a), Char(b))

Exercise

```
fun match (Char a) = CHECK_FOR a
   | match Zero = REJECT
   | match One =  ACCEPT
   | match (Times (r1, r2)) = (match r1) THEN (match r2)
   | match (Plus (r1, r2)) = (match r1) ORELSE (match r2)
   | match (Star r) = REPEAT (match r)
```

accept (Plus(Char(a),Char(b)) “ab”

==> match (Char(a)) ORELSE match (Char(b)) [a,b] List.null

==> (CHECK_FOR a) ORELSE (CHECK_FOR b) [a,b] List.null

==> (**fn** cs1 => …) ORELSE (**fn** cs2 => …) [a,b] List.null

==> (**fn** cs => **fn** k => ((**fn** cs1 => …) cs k) **orelse** ((**fn** cs1 => …) cs k)) [a,b] List.null

==>….

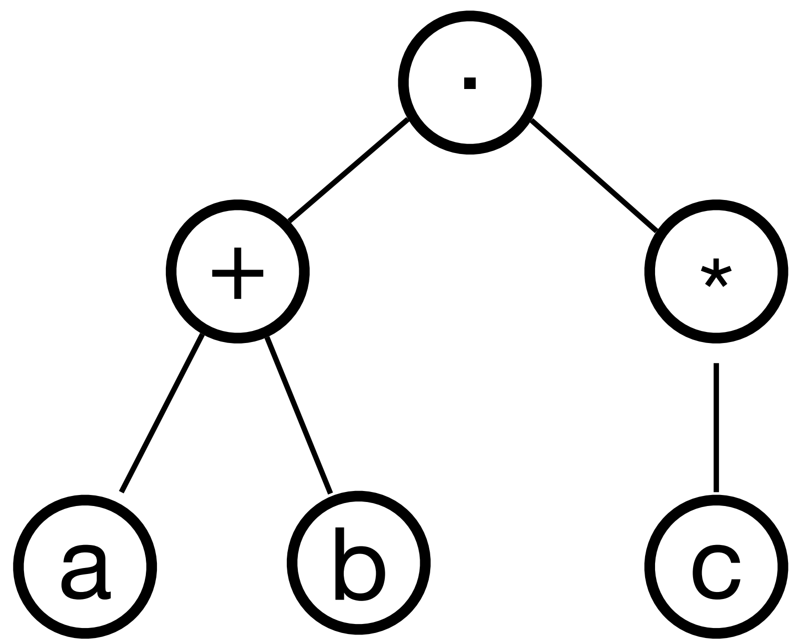
```
==> ((fn cs1 => fn k => (case cs of [ ] => false
                                  | (c::cs') => (a=c) andalso k(cs’)) [a,b] List.null)
                                  orelse
    ((fn cs1=>fn k =>(case cs of [ ] => false
                                  | (c::cs') => (b=c) andalso k(cs’)) [a,b] List.null)
```

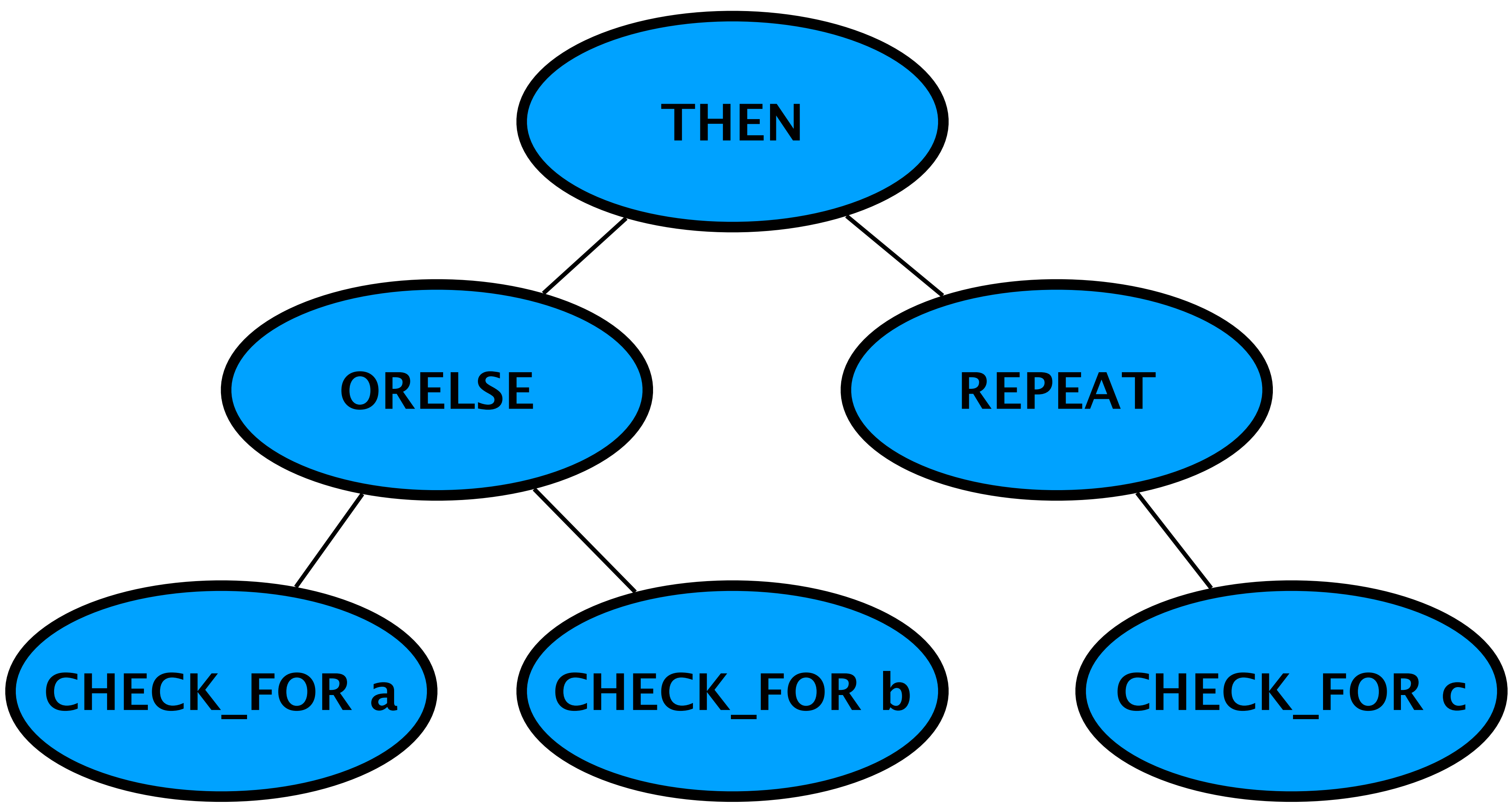
==> false

Build a matcher from a regexp

```
fun match (Char a) = CHECK_FOR a
  | match One = ACCEPT
  | match Zero = REJECT
  | match (Times (r1, r2)) =(match r1) THEN (match r2)
  | match (Plus (r1, r2)) = (match r1) ORELSE (match r2)
  | match (Star r) = REPEAT (match r)
```

(a+b) c*





```
fun match (Char a) = CHECK_FOR a
  | match Zero = REJECT
  | match One = ACCEPT
  | match (Times (r1, r2)) =(match r1) THEN (match r2)
  | match (Plus (r1, r2)) = (match r1) ORELSE (match r2)
  | match (Star r) = REPEAT (match r)
```

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

Staged matcher

```
fun accept (r : regexp) : string -> bool =

    let
        val m = match r
    in
        fn s => m (String.explode s) List.null
    end
```