15-150 Fall 2013
Lecture 13

Stephen Brookes

There are two ways of constructing a software design:

One way is to make it so simple that there are obviously no deficiencies.

The other way is to make it so complicated that there are no obvious deficiencies.

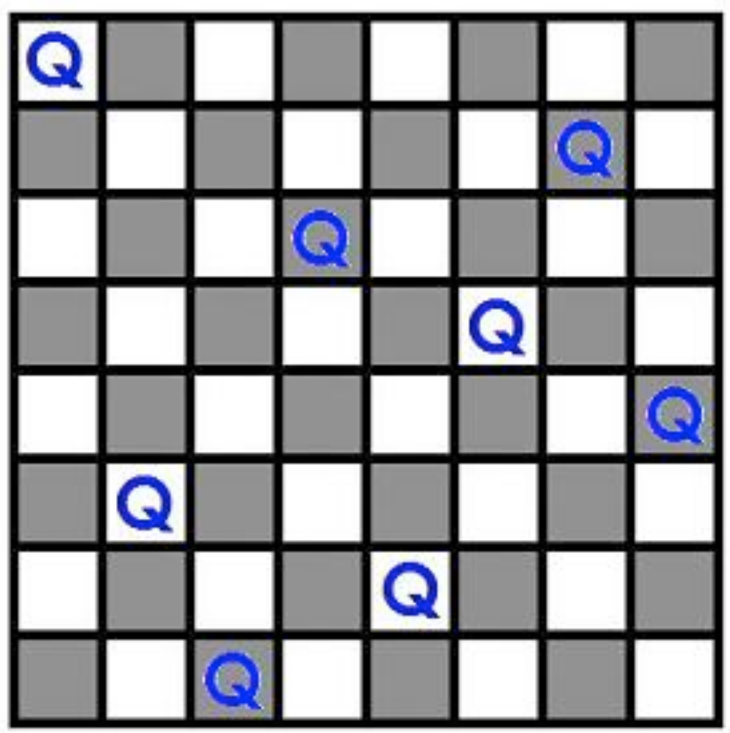
— C.A.R. Hoare, The 1980 ACM Turing Award Lecture

The most effective debugging tool is ... careful thought...

— Brian W. Kernighan, in the paper Unix for Beginners (1979)

n queens

- The royal family's ***favourite*** algorithm



before we start

- Number of ways to put 8 queens on board

$$\frac{64 * 63 * 62 * 61 * 60 * 59 * 58 * 57}{8 * 7 * 6 * 5 * 4 * 3 * 2 * 1} > 4{,}000{,}000{,}000$$

- Number of ways using different columns

$$8 * 7 * 6 * 5 * 4 * 3 * 2 * 1 \quad = 40{,}320$$

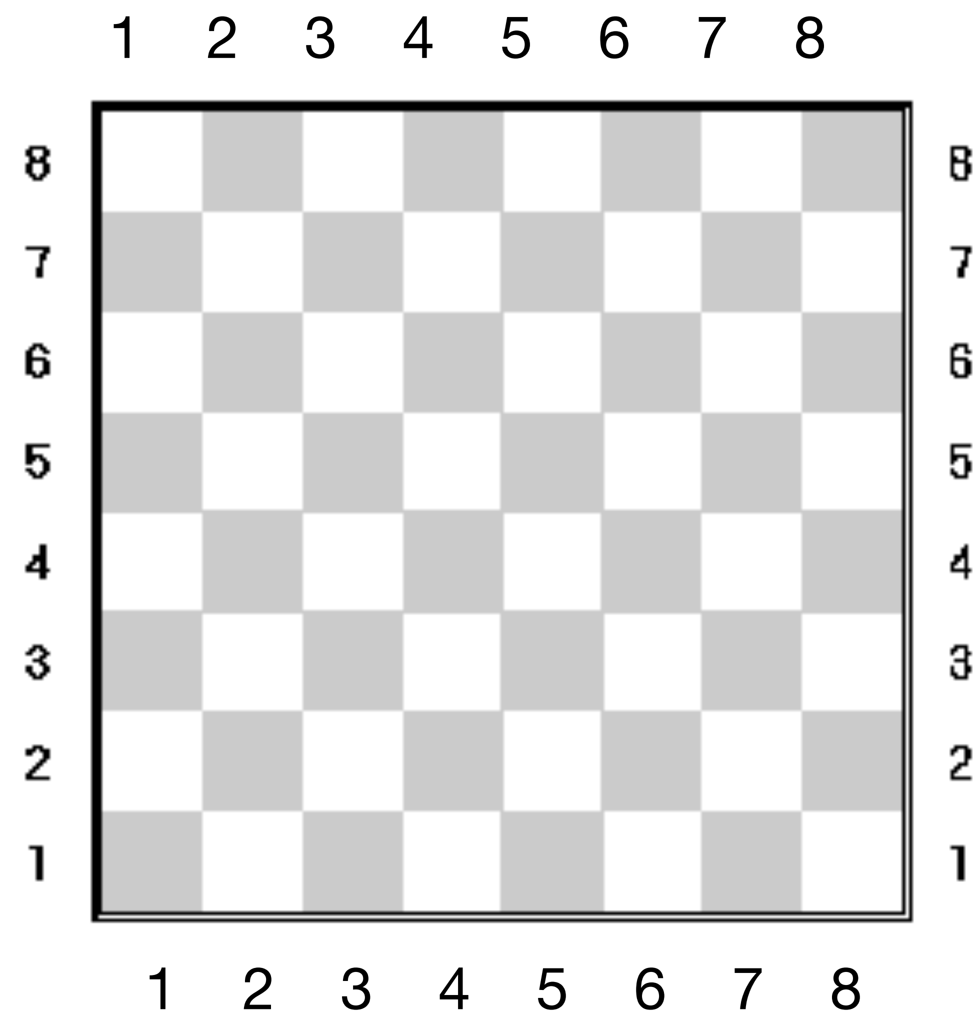
- Number of *safe* ways 92

Fast algorithm for this problem?

PRICELESS

board

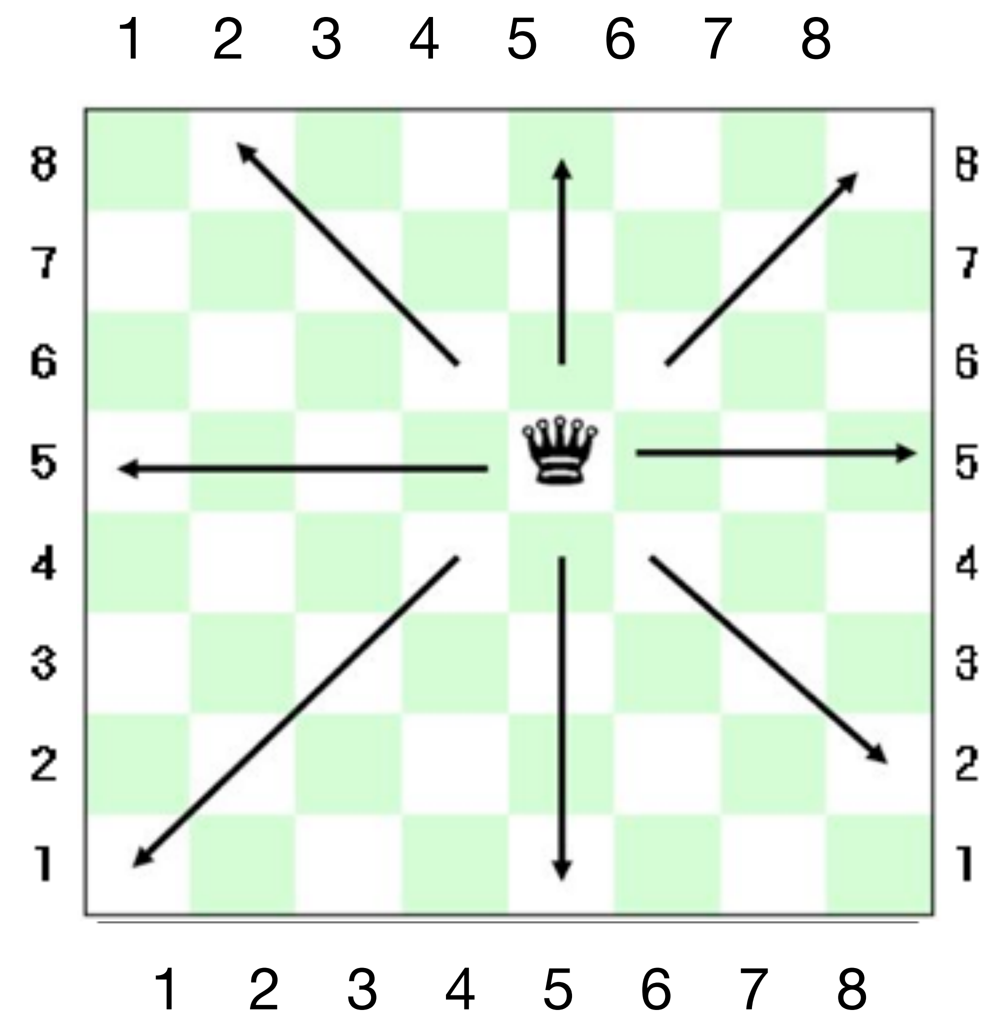
type row = int
type col = int
type pos = row * col
type sol = pos list



threat

threat : pos * pos -> bool

```
fun threat((x,y), (i,j)) =
  (x=i) orelse
    (y=j) orelse
      (x+y = i+j) orelse
        (x-y = i-j)
```



threat(p, q) = **true**
iff p *attacks* q

conflict

(* conflict : pos * pos list -> bool *)

fun conflict (p, nil) = **false**
| conflict (p, q::qs) = threat(p, q) **orelse** conflict(p, qs)

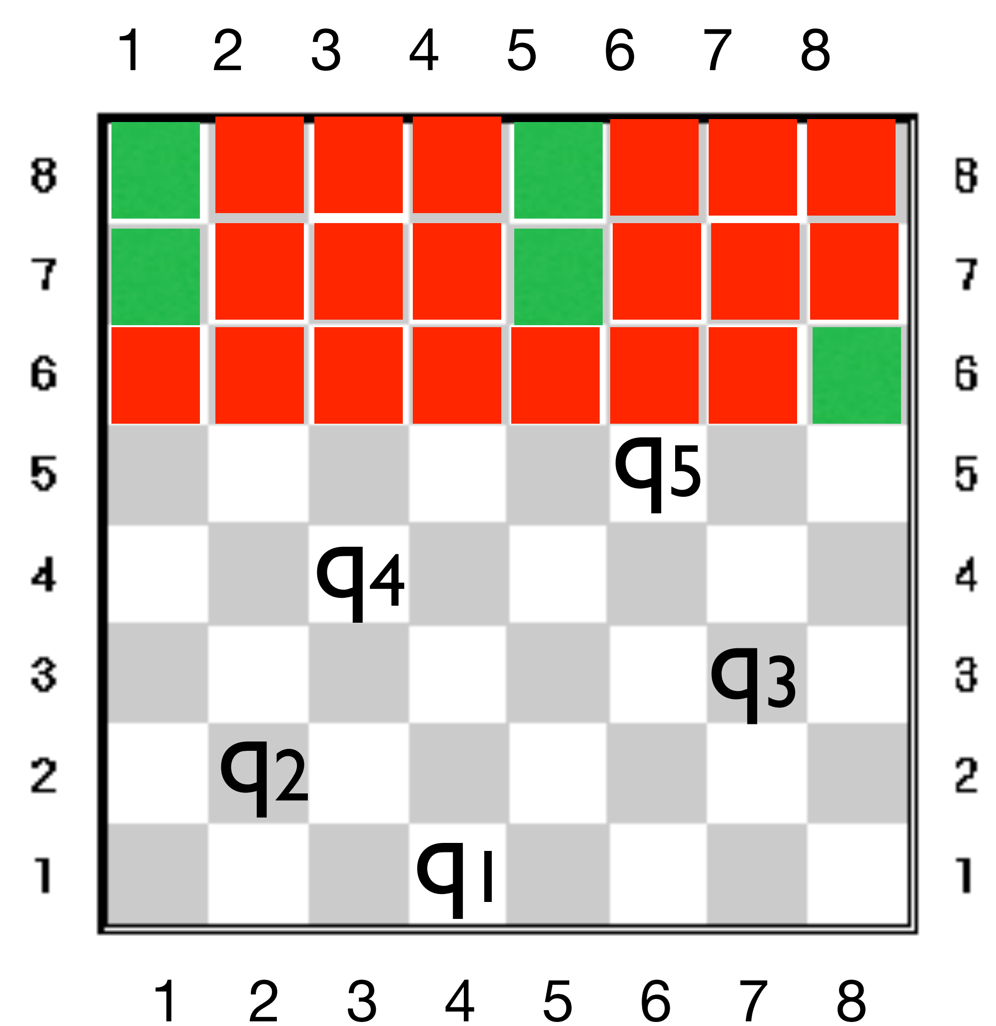
conflict (p, [q_1,...,q_k]) = **false**

if threat(p, q_i) = **false**

for each i

conflict (p, [q_1,...,q_k]) = **true**

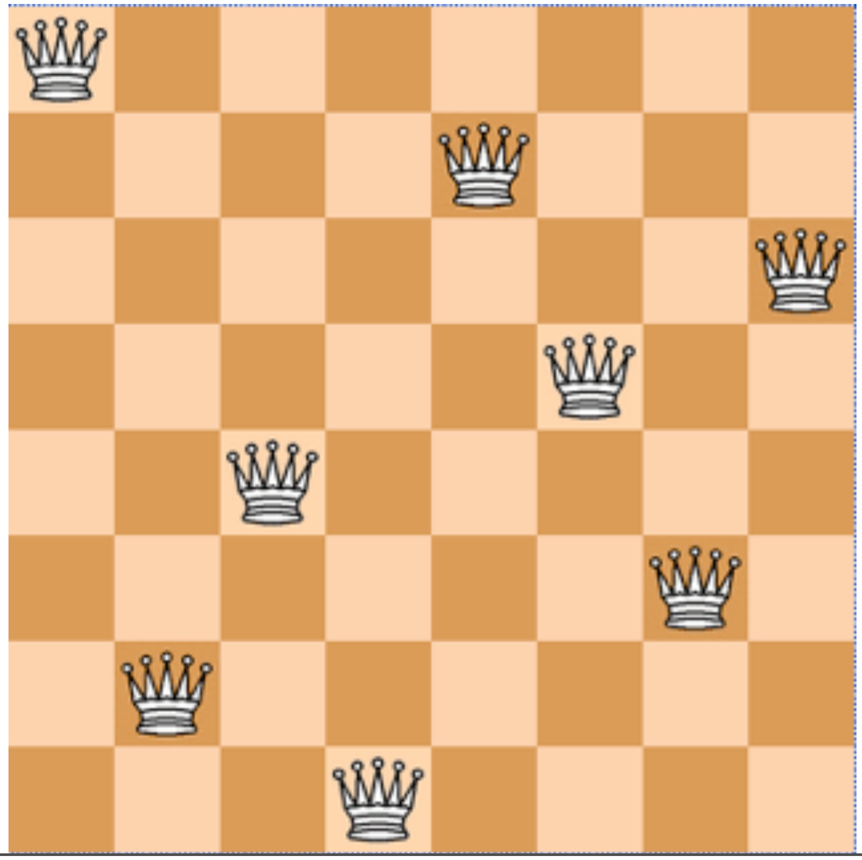
otherwise



safe

safe : pos list -> bool

fun safe [ ] = **true**
| safe (q::qs) = **if** conflict(q, qs)
then false
else safe(qs)



safe $[q_1,...,q_k]$ = **true**
iff for all $i \neq j$,
q_i and q_j are on different
rows, columns and diagonals

solutions

qs is a
partial solution
for rows 1 *through* i
if
safe qs = **true** &
map (**fn** (i, j) => i) qs = [i, ..., 1]

A *full solution* to the n-queens problem
is a partial solution qs for rows 1 through n
such that every (i,j) in qs has 1 ≤ j ≤ n

answers

type ans = sol option

SOME qs means *found solution* qs
NONE means *no solution*

continuations

type cont = unit -> ans

try

try : int * row * col list * sol -> cont -> ans

try(n, i, A, qs) fc

(* REQUIRES: 1 ≤ i ≤ n and A is a sublist of [1,. . .,n]
& qs is a partial solution for rows 1 through i-1 *)

(* ENSURES:
EITHER try (n, i, A, qs) fc = SOME(qs')
where qs' is a full solution extending qs
with a queen in row i at a column in A
OR try (n, i, A, qs) fc = fc( )
& there is no solution extending qs
with a queen in row i at a column in A *)

try(n, i, A, qs) fc

columns to be
tried for row i

partial solution
on rows 1...i-1

If A = [ ], there is no solution extending qs, so **fail**

If A = j::B,
– **either** (i,j) is attacked by qs,
so try columns from B for row i;
– **or** it's safe to extend qs with (i,j);
if i=n, (i,j)::qs is a full solution;
otherwise, look for a solution extending (i,j)::qs in row i+1,
and if this fails, backtrack to try columns from B for row i.

try(n, i, A, qs) fc

columns to be tried for row i

partial solution on rows 1...i-1

If A = [ ], there is no solution extending qs, so **fail**

fc( )

conflict((i,j), qs)=**true**

If A = j::B,

– **either** (i,j) is attacked by qs,
so try columns from B for row i;

try(n, i, B, qs) fc;

– **or** it's safe to extend qs with (i,j);
if i=n, (i,j)::qs is a full solution;
otherwise, look for a solution extending (i,j)::qs in row i+1,
and if this fails, backtrack to try columns from B for row i.

conflict((i,j), qs)=**false**

try(n, i+1, [1,...,n], (i,j)::qs)
(fn ( ) => try(n, i, B, qs) fc)

try

```
fun try (n, i, A, qs) fc =
    case A of
        [ ]  => fc( )
      | j::B => if conflict((i, j), qs)
                then try (n, i, B, qs) fc
                else if i = n
                     then SOME( (i, j)::qs )
                     else
                       try(n, i+1,  upto 1 n, (i, j)::qs)
                           (fn ( ) => try (n, i, B, qs) fc)
```

queens

queens : int -> sol option

```
(* REQUIRES: n>0 *)

(* ENSURES:
     EITHER  queens n = SOME qs
                where qs is an n-queens solution;
     OR         queens n = NONE
                & there is no n-queens solution.   *)
```

queens

Look for solution extending [ ] with a queen in row 1

```
fun queens n =
    try (n, 1, upto 1 n, nil)
            (fn ( ) => NONE);
```

Pessimistic failure continuation

results

queens 3 = NONE

queens 4 = SOME [(4,3),(3,1),(2,4),(1,2)]

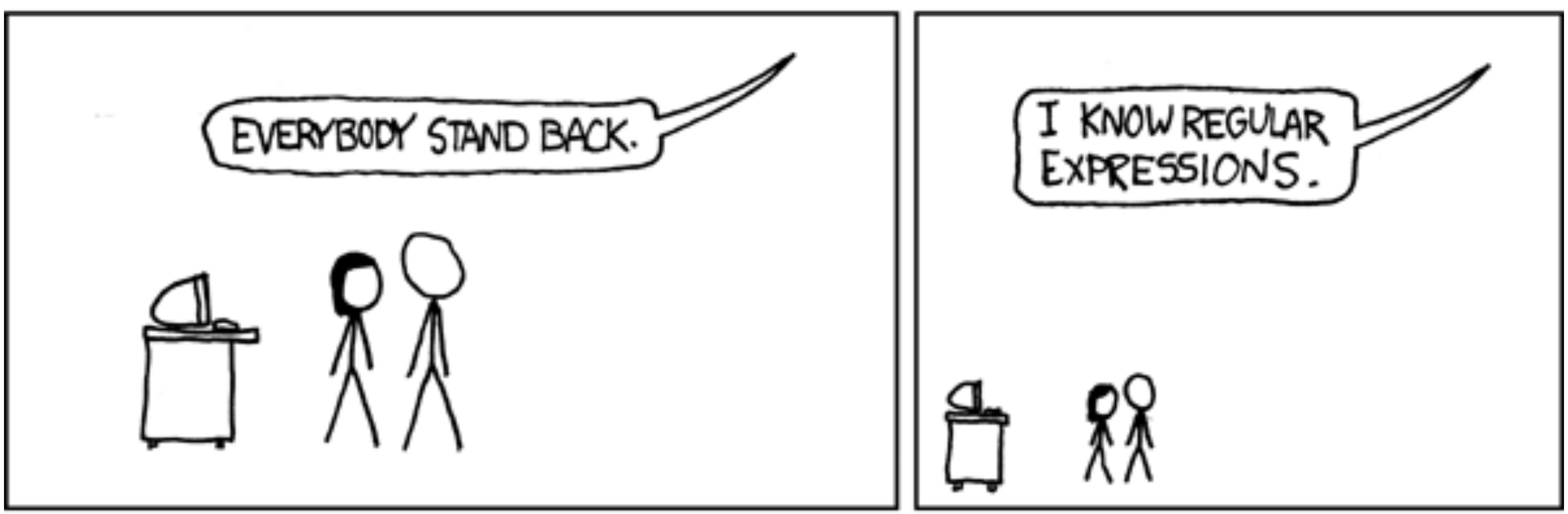
queens 5 = SOME [(5,4),(4,2),(3,5),(2,3),(1,1)]

queens 8 = SOME [(8,4),(7,2),(6,7),(5,3),(4,6),(3,8),(2,5),(1,1)]

queens 20 = SOME [(20,11),(19,6),(18,14),(17,7),(16,10),...]

regular expressions

- using a datatype
- structural induction



regular expressions

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

```
Zero  : regexp
One   : regexp
Char  : char -> regexp
Plus  : regexp * regexp -> regexp
Times : regexp * regexp -> regexp
Star  : regexp -> regexp
```

$$R ::= 0 \mid 1 \mid c \mid R_1{+}R_2 \mid R_1\ R_2 \mid R^*$$

regular expressions

The *values* of type regexp are *inductively* generated by the following rules:

- Zero, One, Char c are values
- If R_1 and R_2 are values, so are

  Plus(R_1, R_2) and Times(R_1, R_2)

- If R is a value, so is Star R

regular languages

A regular expression R

denotes a language $\mathcal{L}$(R) *... a set of char lists*

$\mathcal{L}(\text{Zero}) = \{\ \}$

$\mathcal{L}(\text{One}) = \{\ [\ ]\ \}$

$\mathcal{L}(\text{Char } c) = \{\ [c]\ \}$

$\mathcal{L}(\text{Plus}(R_1,R_2)) = \mathcal{L}(R_1) \cup \mathcal{L}(R_2)$

$\mathcal{L}(\text{Times}(R_1,R_2)) = \{L_1 @ L_2 \mid L_1 \in \mathcal{L}(R_1), L_2 \in \mathcal{L}(R_2)\}$

$\mathcal{L}(\text{Star}(R)) = \{[\ ]\} \cup \{L_1 @ L_2 \mid L_1 \in \mathcal{L}(R), L_2 \in \mathcal{L}(\text{Star } R)\}$

comments

$\mathcal{L}(\text{Star}(R)) = \{[\,]\} \cup \{L_1 @ L_2 \mid L_1 \in \mathcal{L}(R), L_2 \in \mathcal{L}(\text{Star } R)\}$

- This is a *recursive* description of $\mathcal{L}(\text{Star}(R))$
- We interpret it as saying that $\mathcal{L}(\text{Star}(R))$ is the smallest set S satisfying the equation

  $$S = \{[\,]\} \cup \{L_1 @ L_2 \mid L_1 \in \mathcal{L}(R), L_2 \in S\}$$

- This set S is

$$\bigcup \{L_1 @ L_2 @ \ldots @ L_n \mid n \geq 0 \ \& \text{ each } L_i \in \mathcal{L}(R)\}$$

specification

- **Write a function to check if L $\in$ $\mathcal{L}$(R)**

accepts : regexp -> char list -> bool

accepts R L = **true**
iff
L $\in$ $\mathcal{L}$(R)

problem

- Not easy to solve *directly*
- For $\mathrm{Times}(R_1,R_2)$ it's *possible* to *generate-and-test* all splits $L_1@L_2$ of L
- But this can be very costly!
- And what about Star R?

solution

Generalize the problem...

- Does **L** have a *prefix* in $\mathcal{L}$**(R)** with a *suffix* that *satisfies* a ***success*** *condition?*

$$L = [\underbrace{c_1, c_2, \ldots, c_k,}_{\textit{prefix}}\ \underbrace{c_{k+1}, \ldots, c_n}_{\textit{suffix}}]$$

intuition

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

success condition

match R L p = **true**
iff L has a split $L=L_1 @ L_2$ with
$L_1 \in \mathcal{L}(R)$ & $p(L_2)=$**true**

a prefix of L
is in $\mathcal{L}$(R)

the rest of L
satisfies p

generalized problem

- Write an ML function
  match : regexp -> char list ->
  (char list -> bool) -> bool

  such that

  (a) match R L p = **true**
  if there are L_1, L_2 such that
  $L = L_1 @ L_2$ & $L_1 \in \mathcal{L}(R)$ & $p(L_2) =$ **true**

  (b) match R L p = **false**
  otherwise

how that helps

Can then define

```
(* accepts : regexp -> char list -> bool *)
   fun accepts R L = match R L null

(* ENSURES:
       accepts R L = true if L is in ℒ(R)
       accepts R L = false otherwise        *)
```

program design

- Define match by structural induction on R

```
fun match Zero L p        = ...
  | match One L p         = ...
  | match (Char c) L p    = ...
  | match (Plus(R1, R2)) L p
      = (* use match R1 and match R2 *)
    ...
```

base cases

$\mathcal{L}$(Zero) = { }

$\mathcal{L}$(One) = { [ ] }

$\mathcal{L}$(Char c) = { [c] }

- match Zero L p = **false**
- match One L p = p(L)
- match (Char c) L p = **case** L **of**

  [ ] => **false**
  | x::L′ =>
  (c=x) **andalso** p(L′)

Plus

$\mathcal{L}(\text{Plus}(R_1,R_2)) = \mathcal{L}(R_1) \cup \mathcal{L}(R_2)$

- match (Plus(R_1, R_2)) L p =

  (match R_1 L p) **orelse** (match R_2 L p)

Times

$\mathcal{L}(\text{Times}(R_1,R_2)) = \{L_1 @ L_2 \mid L_1 \in \mathcal{L}(R_1), L_2 \in \mathcal{L}(R_2)\}$

- match (Times(R_1,R_2)) L p =
  $\text{match } R_1 \text{ L } (\textbf{fn } \underbrace{\text{L}' \text{ => match } R_2 \text{ L}' \text{ p}})$

success continuation
says what to do
when R_1 match succeeds...

try matching R_2 on the *suffix*