Greedy regular expression matching

Alain Frisch (ENS Paris)

Luca Cardelli (MSR Cambridge)

The matching problem

- Problem = project the structure of a regular expression on a flat sequence.
  - $R = (a * |b)*$
  - $w = a_1 a_2 b_1 b_2 a_3$
  - $\Rightarrow v = [1 : [a_1; a_2];\ \ 2 : b_1;\ \ 2 : b_2;\ \ 1 : [a_3]]$
- The result retains the structure of the regexp and the content of the sequence.
- Result driven by the syntax of regexps $\neq$ automata.
- Issues: efficiency, disambiguation.

Main motivation

- Type-directed native representation of values in XDuce-like languages: E.g.:

  `[ int ]` $\leadsto$ `int`
  `[ int int* ]` $\leadsto$ `struct {int fst; int[] snd;}`

- Advantages over uniform representation:
  - More compact representation, less boxing
  - Fast random access
  - Easier to interface/integrate with other language
- Requires coercion between subtypes.
  - Flatten sequences.
  - Project the structure of the new regexp = matching.

Other applications

- Regexp packages with structured matching semantics.
- Lexer-parser generators.
- Sequence/tree transducers (e.g.: Hosoya’s filters).

Proof of concept

- A regexp iterator extension for C#

```
object[] a = new object[] {1,2,3,4,"abc",4,5,"xyz",6,7,false};
applyregexp(a) (
  (
    ( int  , int                          )*,
    string

  )
  |
  (
    ( int                          )*,
    bool

  )
)*;
```

Proof of concept

▷ A regexp iterator extension for C#

```
object[] a = new object[] {1,2,3,4,"abc",4,5,"xyz",6,7,false};
applyregexp(a) (
  ( { int sum = 0; },
    ( int x, int y, { sum += x*y; } )*,
    string s,
    { System.Console.WriteLine(s + ":" + sum); }
 )
 |
  ( { int sum = 0; },
    ( int x, { sum += x; } )*,
    bool b,
    { System.Console.WriteLine(b + ":" + sum); }
  )
)*;
```

⇝

```
abc:14
xyz:20
False:13
```

Key issue: avoiding backtracking

- Consider the regexp:

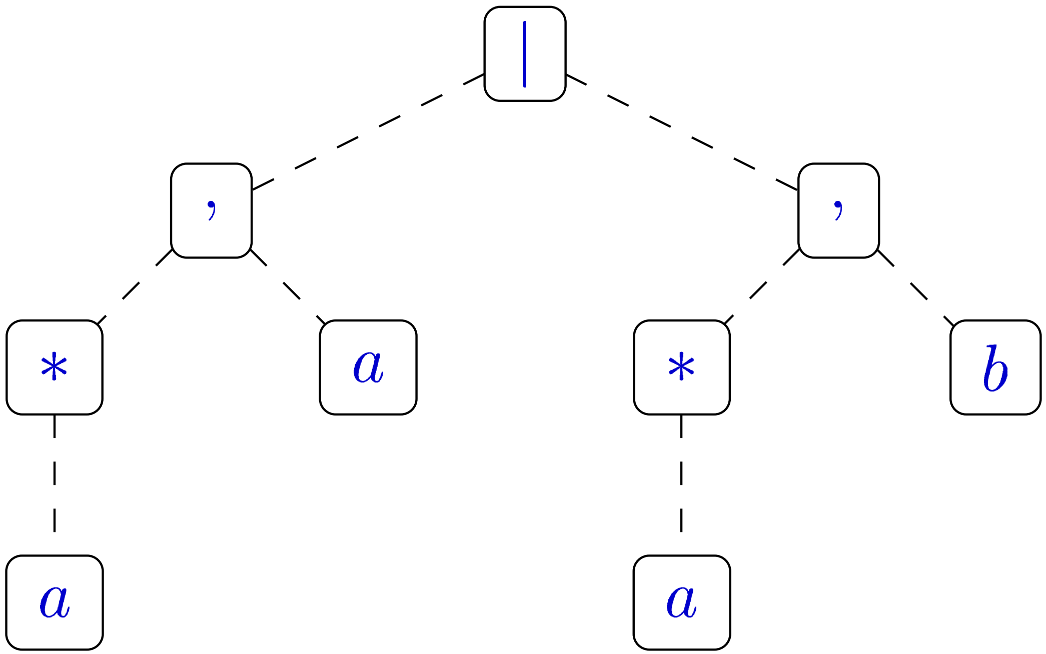
$$R = a* \,|\, (a*, b, a*)$$

- To avoid backtracking, and still proceed by induction on the regexp, we need to decide first which branch to take (left or right?)
- Unbounded look-ahead!

An example

- Abstract syntax tree of the regexp.

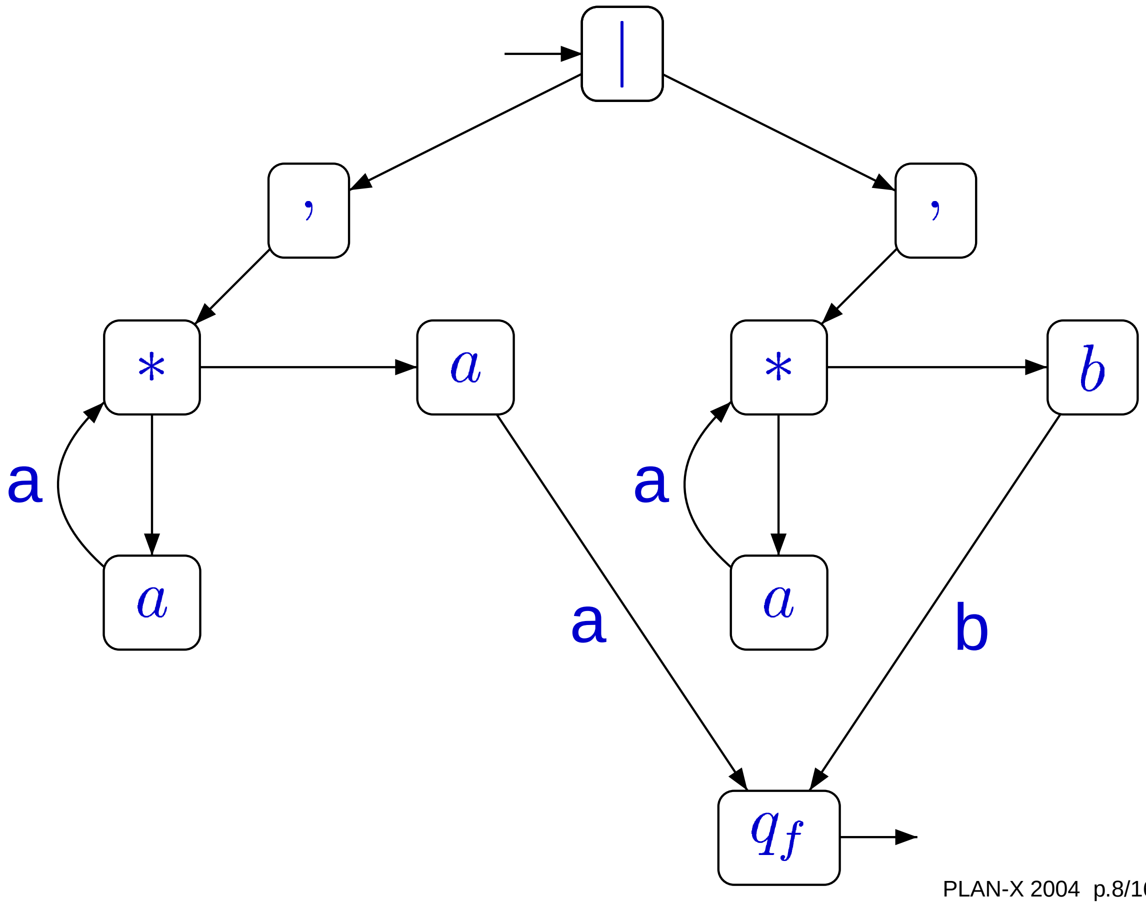
$$R = (a*, a)|(a*, b)$$
$$w = \; a \;\; b$$



An example

- Abstract syntax tree of the regexp.
- Build an automaton.

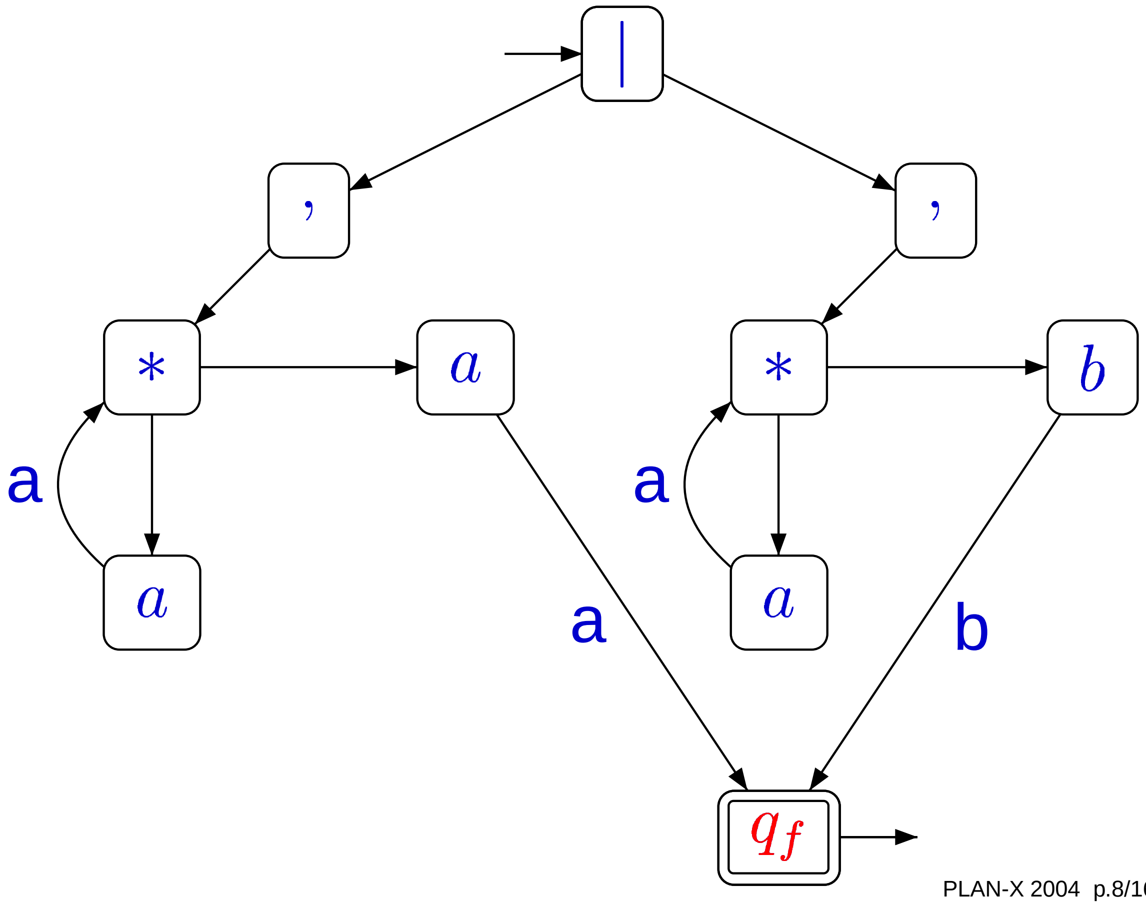
$$R = (a*, a)|(a*, b)$$
$$w = \; a \;\; b$$



An example

- Abstract syntax tree of the regexp.
- Build an automaton.
- Scan the input backwards ("subset construction").

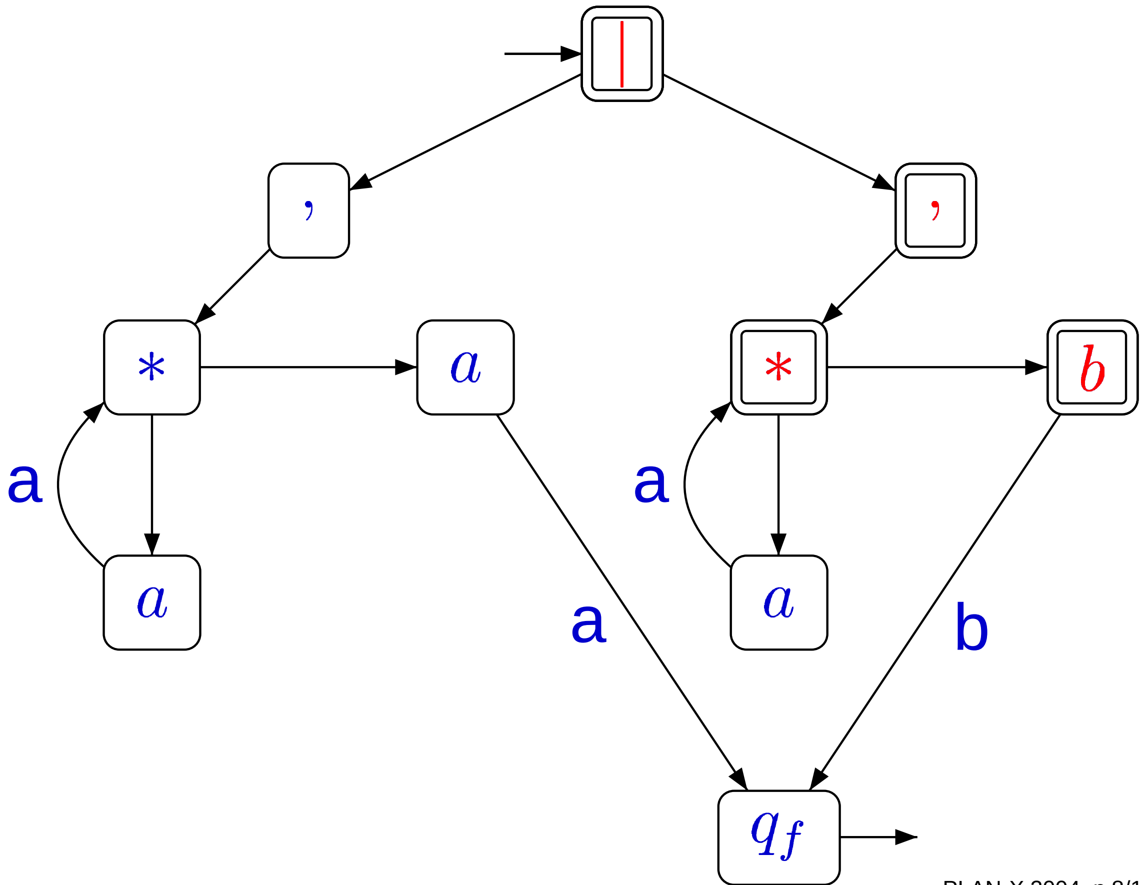
$$R = (a*, a)|(a*, b)$$
$$w = \ a \ \ b$$



An example

- Abstract syntax tree of the regexp.
- Build an automaton.
- Scan the input backwards (“subset construction”).

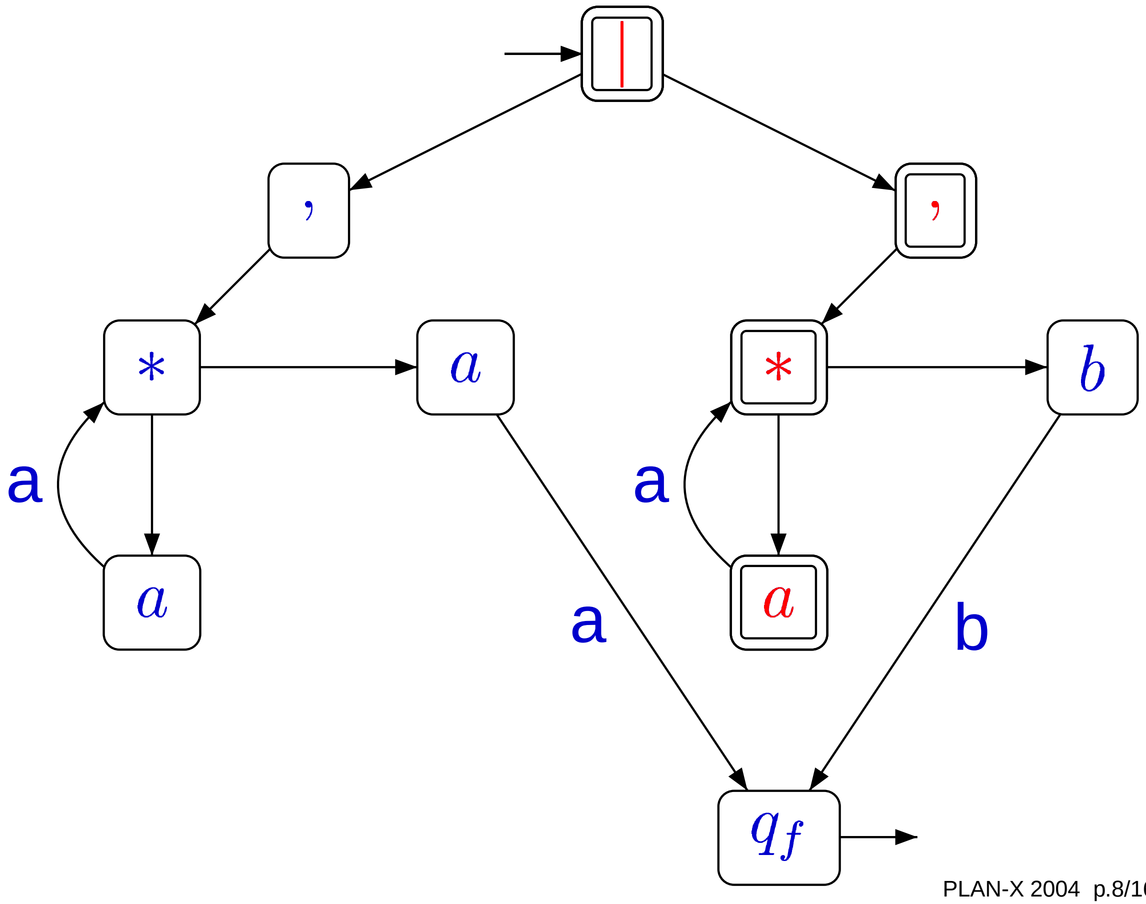
$$R = (a*, a)|(a*, b)$$
$$w = \quad a\bullet b$$



An example

- Abstract syntax tree of the regexp.
- Build an automaton.
- Scan the input backwards ("subset construction").

$$R = (a*, a)|(a*, b)$$
$$w = \bullet a \;\; b$$



Second pass: the matcher

- ```
  let rec loop = function
    | ε -> ()
    | r1 , r2 -> (loop r1, loop r2)
    | r1 | r2 -> if  ... then (1,loop r1) else (2,loop r2)
    | r* -> if  ... then (loop r)::(loop r*) else []
    | c -> (*  Consume the token  *)
  ```
- What are the ... ?
- Given by the first pass.
- Disambiguation:
  - first-match for $|$
  - greedy semantics for $*$
  ```

Non-termination problem

- The algorithm always terminates except with a subregexp $R*$ where R is “nullable”.
- Examples: $(a*, b*)*$ $(a * | b*)*$.
- Same problem in the folklore syntax-directed recognizer:

```
let rec loop r k w = match r with
  | ε -> k w
  | r1 , r2 -> loop r1 (loop r2 k) w
  | r1 | r2 -> (loop r1 k w) || (loop r2 k w)
  | r* -> (loop r (loop r* k) w) || (k w)
  | c -> (w <> []) && (hd w = c) && (k (tl w))

let accept r = loop r ( (=) [] )
```

r : regexp

k : continuation

w : input sequence

`loop` r `k w = true` $\iff$ `w = w1 @ w2` s.t. (r matches `w1`) && (`k w2 = true`)

Non-termination problem

- Simple solution: rewrite regexps to avoid the problematic situation.
  - E.g.: $(a*, b*)* \rightsquigarrow ((a*, b+)|a+)*$
  - The structure of the regexp is lost: not suitable for the matching problem.
  - Interaction with the disambiguation policy ?
- Prevent iterations in stars from accepting empty sequences.
  - In the functional recognizer, replace

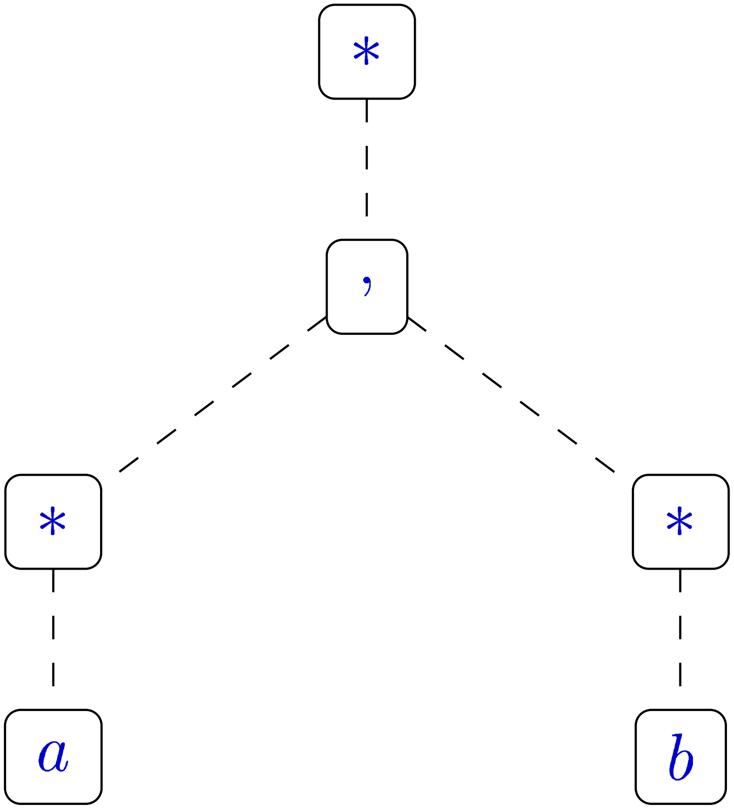
    `(loop` r `(loop` $r*$ `k) w) || (k w)`

    with:

    `(loop` r `(fun w' -> (w <> w') && (loop` $r*$ `k w') w)) || (k w)`
  - How to adapt our algorithm ?

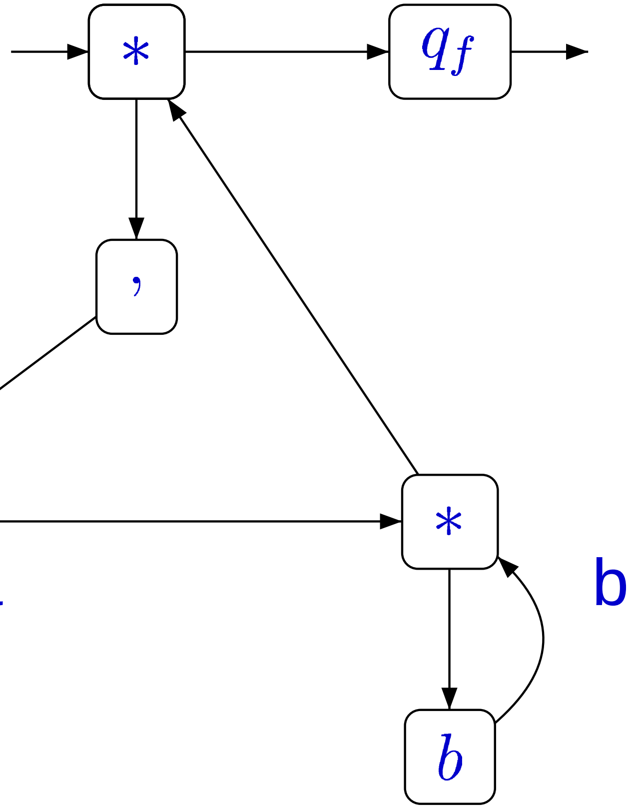
An example

- $R = (a*, b*)*$



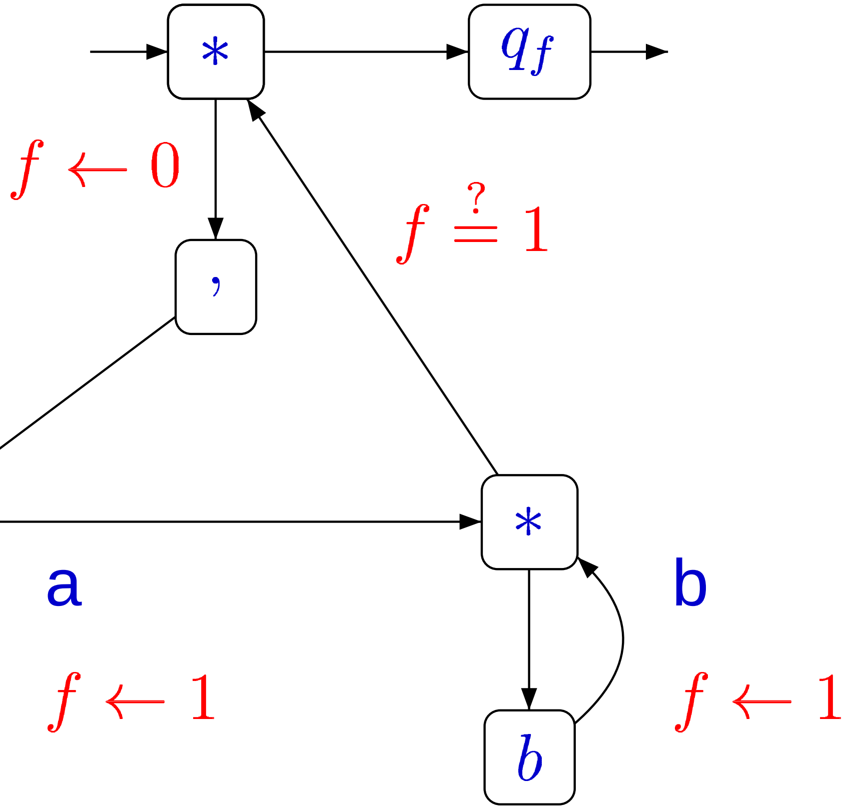
An example

- $R = (a*, b*)*$
- Loop of ε-transitions



An example

- $R = (a*, b*)*$
- Loop of ε-transitions ... now broken.
- Still a fi nite state automaton (states (q, f)).



Second pass: the matcher

▷

```
let rec loop = function
  | ε -> ()
  | r₁ , r₂ -> (loop r₁, loop r₂)
  | r₁ | r₂ -> if  ... then (1,loop r₁) else (2,loop r₂)
  | r* -> if  ... then (f := 0; (loop r)::(loop r*)) else []
  | c -> f := 1;  (* Consume the token *)
```

▷ The ... are given by the first pass.

Summary

- Keep the connection between regexps and automata.
  - Direct construction of the automaton
- Accept problematic regexps, reject problematic matchings.
- Result: linear time (two-passes) matching algorithm, which can be efficiently implemented (bit sets).
- Abstract specification of the disambiguation policy as an optimization problem (not presented).

Future work

- Try and evaluate an alternative implementation technique for Xtatic (use native CLR representations and “value types”).
- Optimizations: the first pass is not always necessary. Use (bounded) look-ahead as long as possible.
- Longest match semantics ?

Thank you!