Compiling
Pattern Matching to
Good Decision Trees

Luc Maranget — Inria Rocquencourt

Motivation

Two targets for match compilers.

- Backtracking automata (Augustsson FPCA'85, OCaml, Haskell).
- Decision trees (SML).

Benefits/drawbacks:

- Backtracking automata offer linear guarantee in code size, but may test the same position more than once.
- Decision trees are just the opposite.

A matter of code size vs. runtime efficiency?

A practical approach

Compare optimizing match compiler experimentally.

- Compiling to decision trees.
- Good decision trees.
- Compiling to good decision trees.
- Compare.

Reference for backtracking automata: OCaml.

Compiling pattern matching?

A (Ca)ML program.

```
type bool = T | F

let f x y z =  match x,y,z with
| _,F,T -> 1
| F,T,_ -> 2
| _,_,F -> 3
| _,_,T -> 4
```

Compile:

- Write the same, without using `match` – here using `if`.

To decision trees:

- When testing `x`, `y` or `z`, consider all possible values, and draw all conclusions.

Matrices, the right tool

Simultaneous matching of `x`, `y`, `z`.

$$\begin{pmatrix} \texttt{x} & \texttt{y} & \texttt{z} \end{pmatrix}$$

$$\begin{pmatrix} \_ & \texttt{F} & \texttt{T} \rightarrow 1 \\ \texttt{F} & \texttt{T} & \_ \rightarrow 2 \\ \_ & \_ & \texttt{F} \rightarrow 3 \\ \_ & \_ & \texttt{T} \rightarrow 4 \end{pmatrix}$$

Test variables, one by one

Let us start by testing `x`.

`x` can be `F`, or something else (look at constructors in column `x`).

Assume x is F

Then, $\texttt{y}$ and $\texttt{z}$ are still to be tested.

$$\begin{matrix} \begin{pmatrix} \texttt{x} & \texttt{y} & \texttt{z} \end{pmatrix} \\ \begin{pmatrix} \_ & \texttt{F} & \texttt{T} \rightarrow 1 \\ \texttt{F} & \texttt{T} & \_ \rightarrow 2 \\ \_ & \_ & \texttt{F} \rightarrow 3 \\ \_ & \_ & \texttt{T} \rightarrow 4 \end{pmatrix} \end{matrix} \qquad\qquad \begin{matrix} \begin{pmatrix} \texttt{y} & \texttt{z} \end{pmatrix} \\ \begin{pmatrix} \texttt{F} & \texttt{T} \rightarrow 1 \\ \texttt{T} & \_ \rightarrow 2 \\ \_ & \texttt{F} \rightarrow 3 \\ \_ & \texttt{T} \rightarrow 4 \end{pmatrix} \end{matrix}$$

In paper: specialization by constructor $\texttt{F}$.

Assume x is something else

Then, $\mathtt{y}$ and $\mathtt{z}$ are still to be tested.

$$\begin{pmatrix} \mathtt{x} & \mathtt{y} & \mathtt{z} \end{pmatrix} \qquad \qquad \begin{pmatrix} \mathtt{y} & \mathtt{z} \end{pmatrix}$$

$$\begin{pmatrix} \_ & \mathtt{F} & \mathtt{T} & \rightarrow 1 \\ \mathtt{F} & \mathtt{T} & \_ & \rightarrow 2 \\ \_ & \_ & \mathtt{F} & \rightarrow 3 \\ \_ & \_ & \mathtt{T} & \rightarrow 4 \end{pmatrix} \qquad \qquad \begin{pmatrix} \mathtt{F} & \mathtt{T} & \rightarrow 1 \\ \_ & \mathtt{F} & \rightarrow 3 \\ \_ & \mathtt{T} & \rightarrow 4 \end{pmatrix}$$

Row 2 cannot match, by “something else” hypothesis.

In paper: compute default matrix.

First compilation step

$$\mathcal{C}((\mathtt{x}\ \mathtt{y}\ \mathtt{z}), \begin{pmatrix} \_ & \mathtt{F} & \mathtt{T} & \rightarrow 1 \\ \mathtt{F} & \mathtt{T} & \_ & \rightarrow 2 \\ \_ & \_ & \mathtt{F} & \rightarrow 3 \\ \_ & \_ & \mathtt{T} & \rightarrow 4 \end{pmatrix}) \qquad =$$

$$\mathtt{if\ x\ then}\ \mathcal{C}((\mathtt{y}\ \mathtt{z}), \begin{pmatrix} \mathtt{F} & \mathtt{T} & \rightarrow 1 \\ \_ & \mathtt{F} & \rightarrow 3 \\ \_ & \mathtt{T} & \rightarrow 4 \end{pmatrix})\ \mathtt{else}\ \mathcal{C}((\mathtt{y}\ \mathtt{z}), \begin{pmatrix} \mathtt{F} & \mathtt{T} & \rightarrow 1 \\ \mathtt{T} & \_ & \rightarrow 2 \\ \_ & \mathtt{F} & \rightarrow 3 \\ \_ & \mathtt{T} & \rightarrow 4 \end{pmatrix})$$

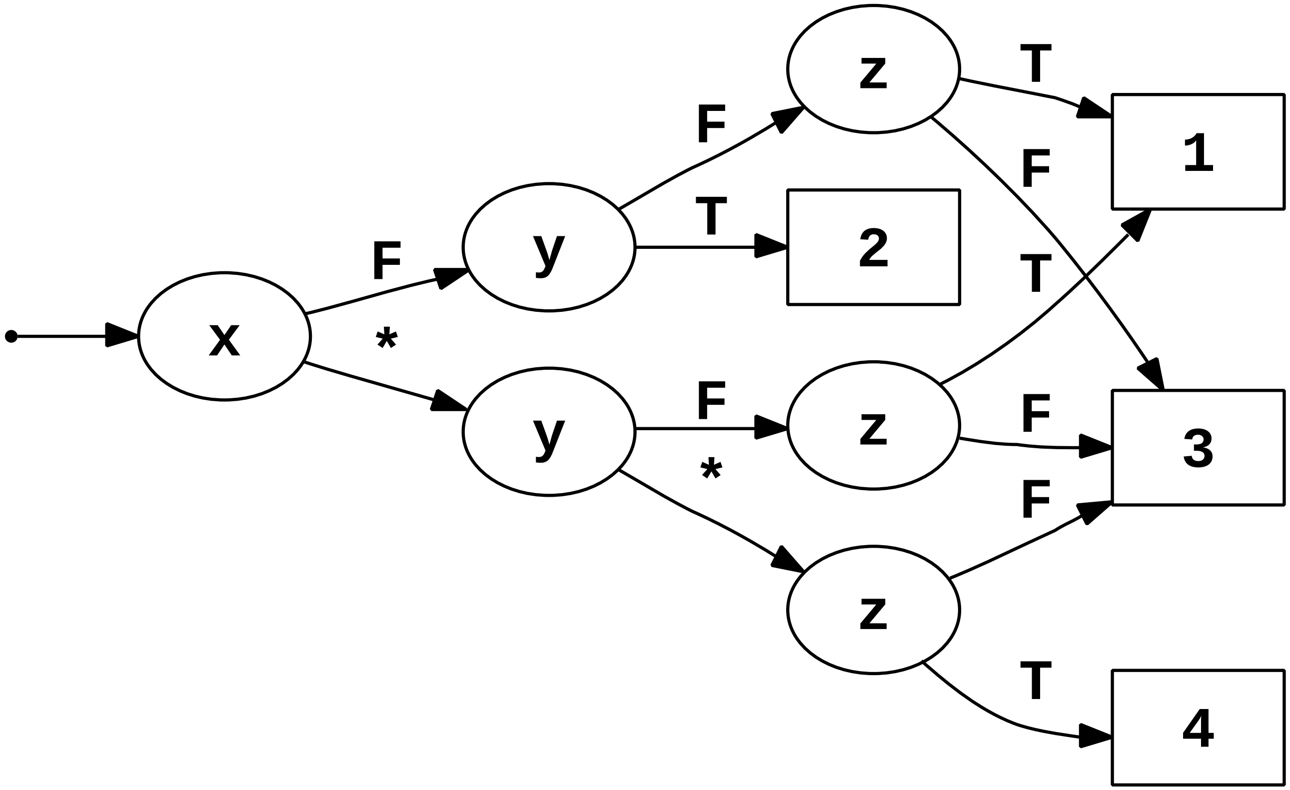
Notice: rows 1, 3 and 4 are duplicated (wildcards).

Output of “naive” compilation

Naive is (depth-first) left-to-right.

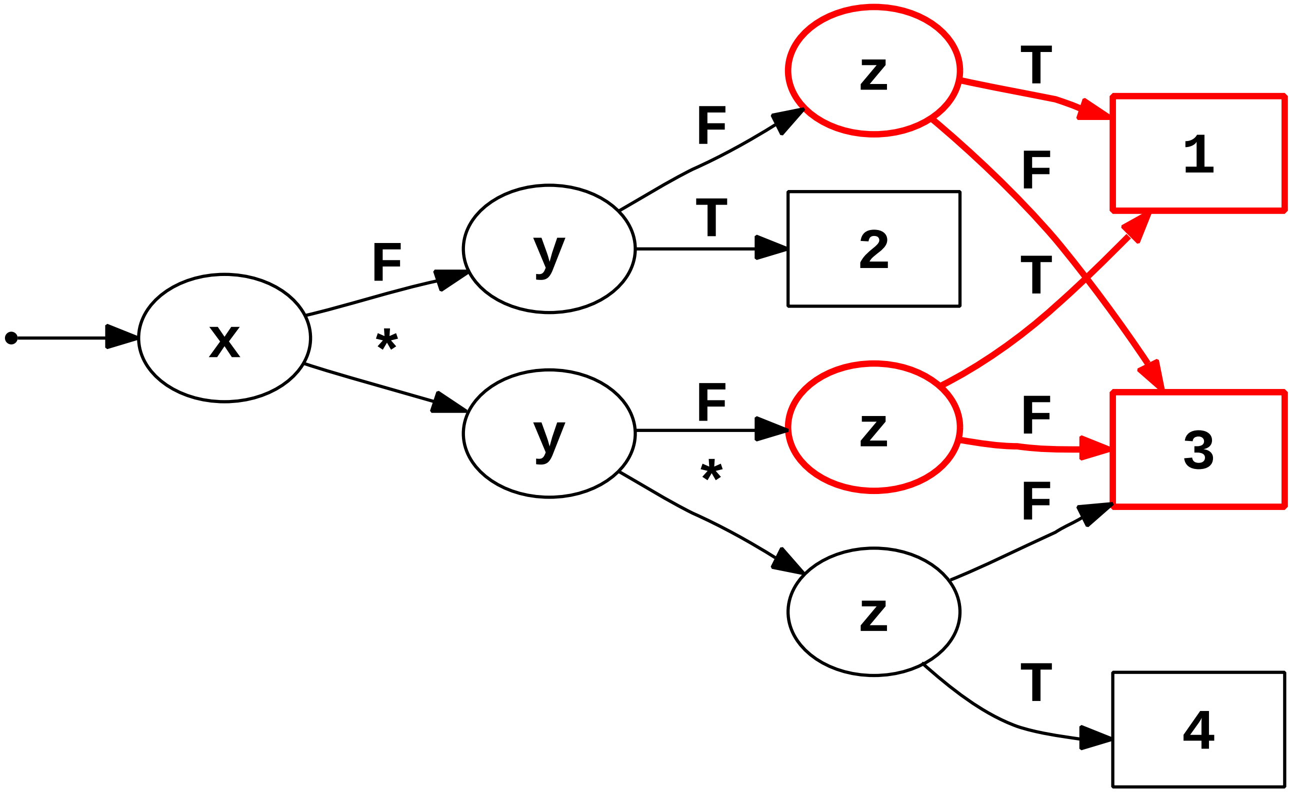
```
let f x y z =
  if x then
    if y then
      if z then 4 else 3
    else
      if z then 1 else 3
  else
    if y then 2
    else
      if z then 1 else 3
```

A better way to show compiler output

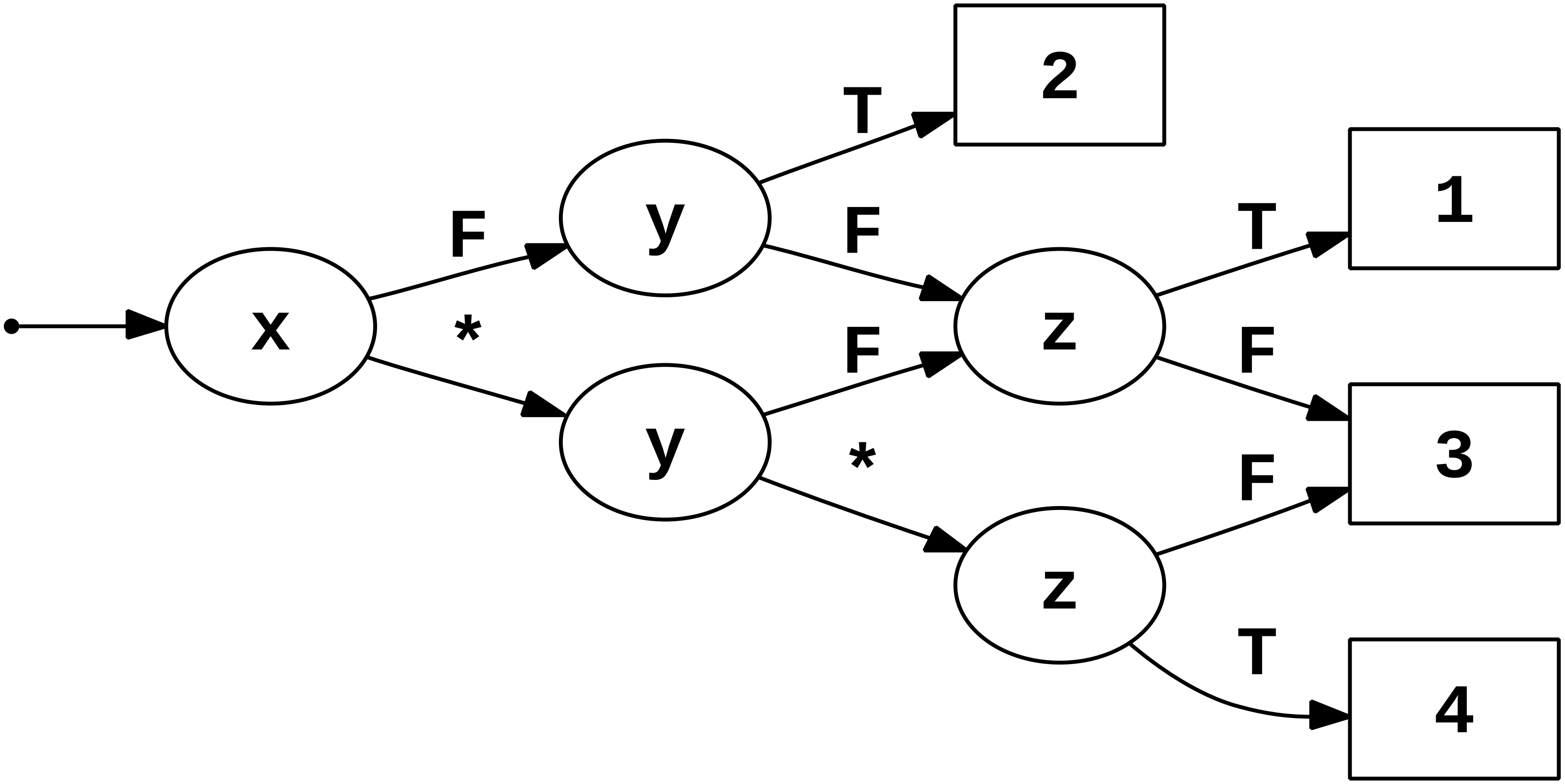


A decision tree, indeed.

Sharing tests



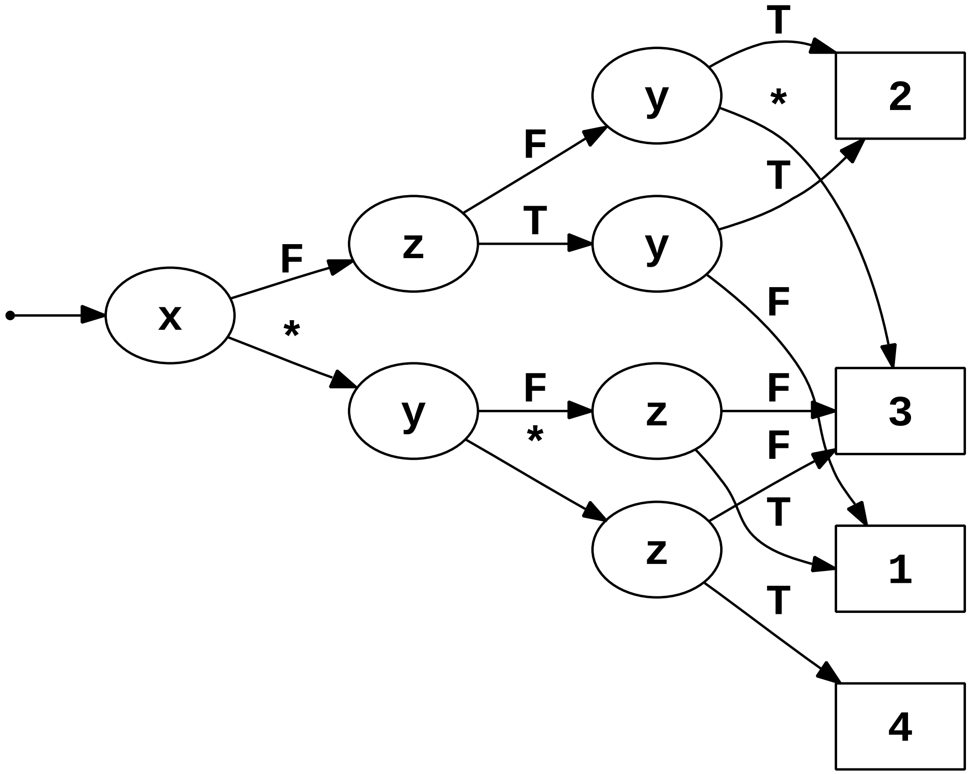
Good decision trees. . . are DAGs



(See Pettersson CC’92)

Why test `x`, `y`, `z` in order ?

Another decision tree.



All of them.

A simpler example

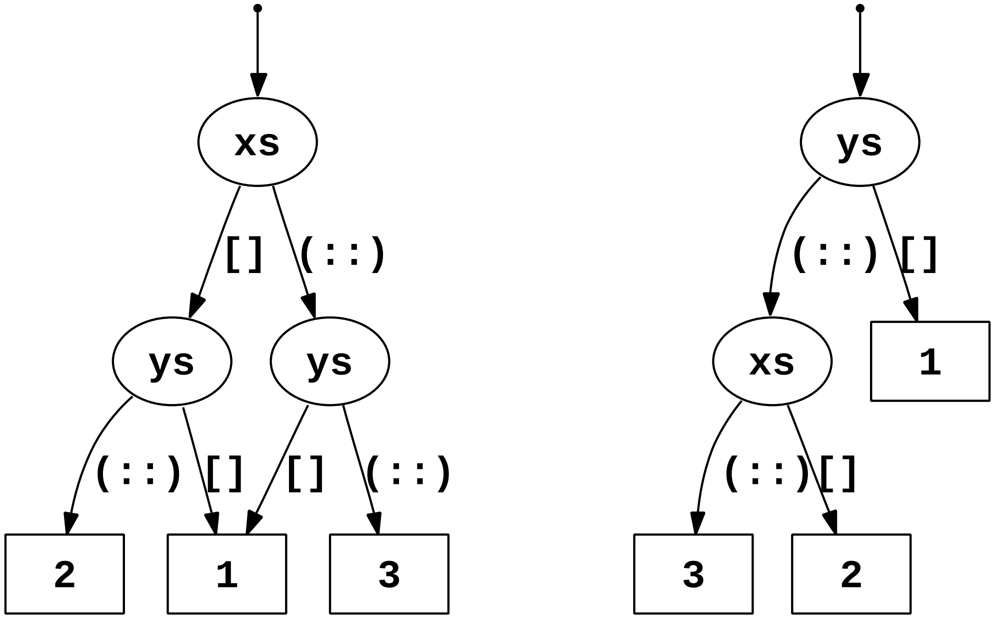
Classical list-merge

```
match xs,ys with
| _,[] -> xs
| [],_ -> ys
| x::xs,y::ys -> ...
```

As a matrix:

$$\begin{pmatrix} \_ & \texttt{[]} & \rightarrow 1 \\ \texttt{[]} & \_ & \rightarrow 2 \\ \_::\_ & \_::\_ & \rightarrow 3 \end{pmatrix}$$

Good decisions trees?



Tree on the right is better.

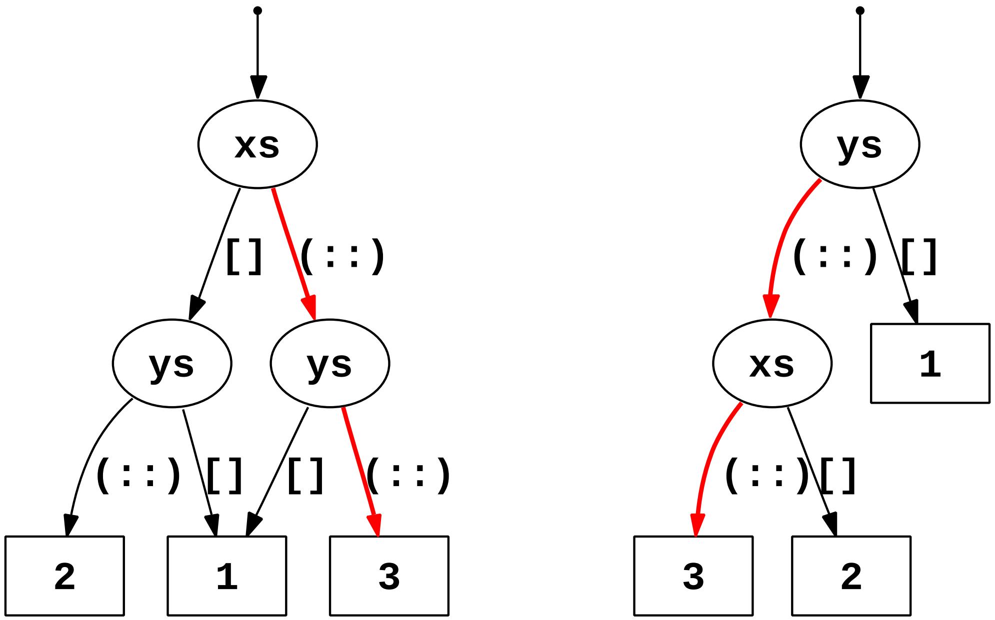
- Smaller: 2 test (switch) nodes vs. 3.
- Shorter path:

  In cases where $ys = $ [] ,we have 1 test vs. 2.

  Runtime behavior identical in other cases.

Necessity

Definition: In matrix (p_i^j), column i is needed for row j when all paths to leaf j test i.



Row 1: `xs` not needed, `ys` needed.

Row 2: `xs` and `ys` needed.

Row 3: `xs` and `ys` needed.

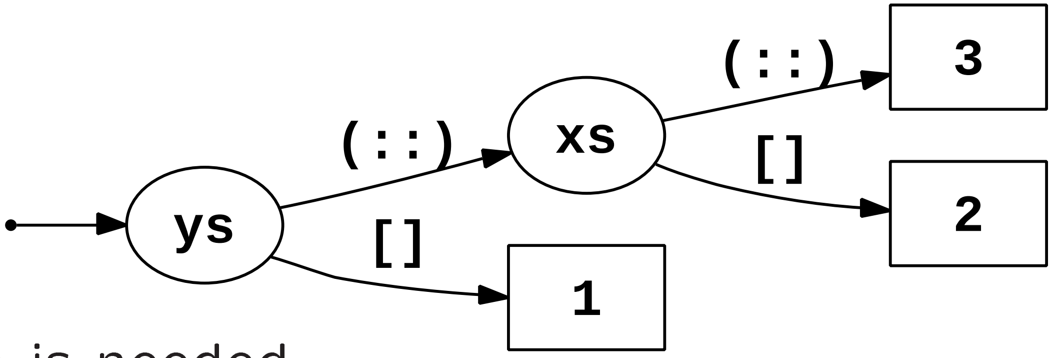
Needed column

Definition: In matrix (p_i^j), column i is needed, when needed for all rows.

Necessity summary as a matrix.

$$\begin{pmatrix} \_ & [] & \rightarrow 1 \\ [] & \_ & \rightarrow 2 \\ \_::\_ & \_::\_ & \rightarrow 3 \end{pmatrix} \qquad \begin{pmatrix} & \bullet \\ \bullet & \bullet \\ \bullet & \bullet \end{pmatrix}$$

An explanation for decision tree quality:



Column `ys` is needed.

Testing needed columns first

Means:

> First test columns that must be tested anyway.

Obviously.

- Tends to yield shorter paths (runtime efficiency).
- Trees with shorter paths are likely to be smaller (code size).

Natural questions

- How to compute necessity?
- What to do when none or several needed columns exist?

Computing necessity on matrices

- If $p_i^j \neq \_$, then column i is needed for row j.
- If $p_i^j = \_$, column i is needed for row j, iff. . .

  Proposition: Row j is useless (redundant) in matrix P with column i erased.

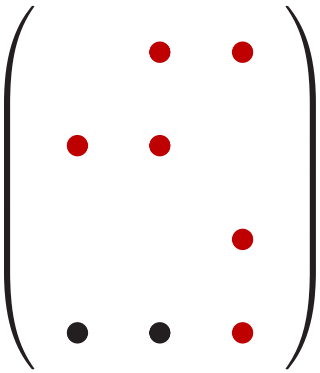
$$P = \begin{pmatrix} \_ & [\,] \\ [\,] & \_ \\ \_::\_ & \_::\_ \end{pmatrix}, \qquad P/1 = \begin{pmatrix} [\,] \\ \_ \\ \_::\_ \end{pmatrix}, \qquad P/2 = \begin{pmatrix} \_ \\ [\,] \\ \_::\_ \end{pmatrix}$$

First row of $P/\texttt{xs}$ useful: $\texttt{xs}$ not needed for row 1.

Second row of $P/\texttt{ys}$ useless: $\texttt{ys}$ needed for row 2.

Computing necessity, more difficult

$$\begin{pmatrix} \_ & \mathtt{F} & \mathtt{T} & \rightarrow 1 \\ \mathtt{F} & \mathtt{T} & \_ & \rightarrow 2 \\ \_ & \_ & \mathtt{F} & \rightarrow 3 \\ \_ & \_ & \mathtt{T} & \rightarrow 4 \end{pmatrix}$$



- Constructor patterns.
- Wildcards..

Heuristics

$$\begin{array}{c} \begin{pmatrix} \mathtt{x} & \mathtt{y} & \mathtt{z} \end{pmatrix} \\ \begin{pmatrix} \_ & \mathtt{F} & \mathtt{T} & \rightarrow 1 \\ \mathtt{F} & \mathtt{T} & \_ & \rightarrow 2 \\ \_ & \_ & \mathtt{F} & \rightarrow 3 \\ \_ & \_ & \mathtt{T} & \rightarrow 4 \end{pmatrix} \end{array} \qquad \begin{array}{c} \begin{pmatrix} \mathtt{x} & \mathtt{y} & \mathtt{z} \end{pmatrix} \\ \begin{pmatrix} & \bullet & \bullet \\ \bullet & \bullet & \\ & & \bullet \\ \bullet & \bullet & \bullet \end{pmatrix} \end{array}$$

- Heuristic n, needed columns. Favor columns needed for a maximal number of rows (here $\mathtt{y}$ and $\mathtt{z}$).
- Heuristic p, needed prefix. Favor columns needed for a maximal prefix of rows (here $\mathtt{y}$).

Approximations in heuristics

Replace “column i is needed for row j” by “p_i^j is a constructor pattern”.

1. Avoid (complex, costly) necessity computations.
2. Avoid copying rows into all specialized matrices.

- Heuristic d, small default. Approximation of n.
- Heuristic q, constructor prefix. Approximation of p.
- Heuristic f, first row. Favor columns whose first pattern is a constructor pattern — radical approximation of p.

d and f were previously known (Scott & Ramsey 2000, SML)

Other heuristics

- Heuristic b, small branching factor. Favor columns with a minimal number of different head constructors. Etc.
- A total of 9 simple heuristics p, n, d, q, f, b a, ℓ, r.
- Sequences of heuristics pb. . . .
- Pseudo-heuristics are total ordering over subterms. L (breadth-first, left-to-right) , R (breadth-first, right-to-left) and N (naive compilation).

Effect of heuristics

$$\begin{pmatrix} \_ & \mathtt{F} & \mathtt{T} \rightarrow 1 \\ \mathtt{F} & \mathtt{T} & \_ \rightarrow 2 \\ \_ & \_ & \mathtt{F} \rightarrow 3 \\ \_ & \_ & \mathtt{T} \rightarrow 4 \end{pmatrix} \qquad \begin{pmatrix} & \bullet & \bullet \\ \bullet & \bullet & \\ & & \bullet \\ \bullet & \bullet & \bullet \end{pmatrix}$$

Heuristic nR selects $\mathtt{z}$ (n$\{\mathrm{x},\mathrm{y},\mathrm{z}\} \rightarrow \{\mathrm{y},\mathrm{z}\}$, R$\{\mathrm{y},\mathrm{z}\} \rightarrow \mathrm{z}$).

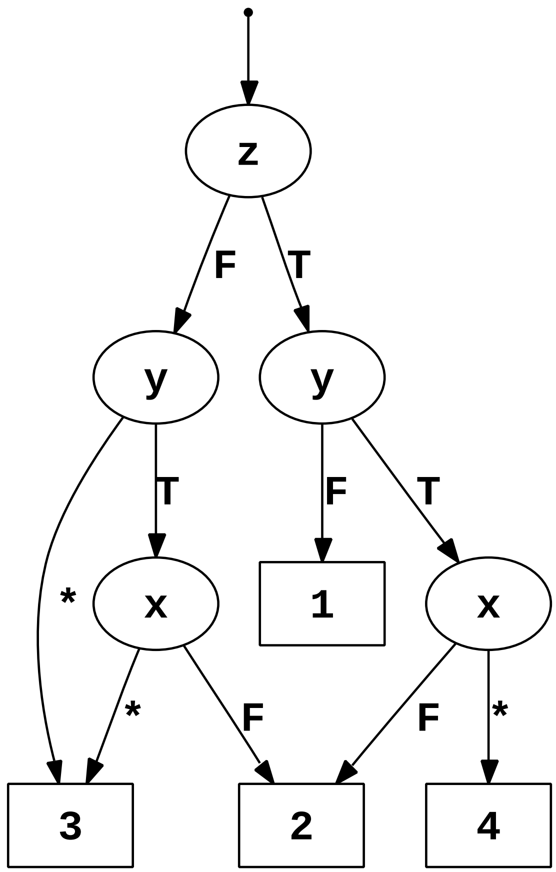
$$\mathtt{if}\ \mathtt{z}\ \mathtt{then}\ \mathcal{C}((\mathrm{x}\ \ \mathrm{y}), \begin{pmatrix} \_ & \mathtt{F} & \rightarrow & 1 \\ \mathtt{F} & \mathtt{T} & \rightarrow & 2 \\ \_ & \_ & \rightarrow & 4 \end{pmatrix}\ \mathtt{else}\ \mathcal{C}((\mathrm{x}\ \ \mathrm{y}), \begin{pmatrix} \mathtt{F} & \mathtt{T} & \rightarrow & 2 \\ \_ & \_ & \rightarrow & 3 \end{pmatrix}$$

Heuristic p selects $\mathtt{y}$ (p$\{\mathrm{x},\mathrm{y},\mathrm{z}\} \rightarrow \mathrm{y}$).

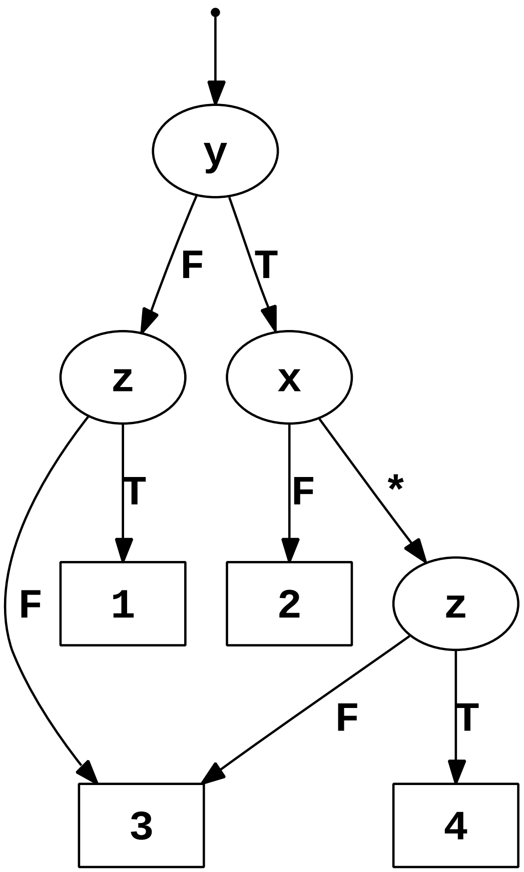
$$\mathtt{if}\ \mathtt{y}\ \mathtt{then}\ \mathcal{C}((\mathrm{x}\ \ \mathrm{z}), \begin{pmatrix} \mathtt{F} & \_ & \rightarrow & 2 \\ \_ & \mathtt{F} & \rightarrow & 3 \\ \_ & \mathtt{T} & \rightarrow & 4 \end{pmatrix}\ \mathtt{else}\ \mathcal{C}((\mathrm{x}\ \ \mathrm{z}), \begin{pmatrix} \_ & \mathtt{T} & \rightarrow & 1 \\ \_ & \mathtt{F} & \rightarrow & 3 \\ \_ & \mathtt{T} & \rightarrow & 4 \end{pmatrix}$$

And finally

nR

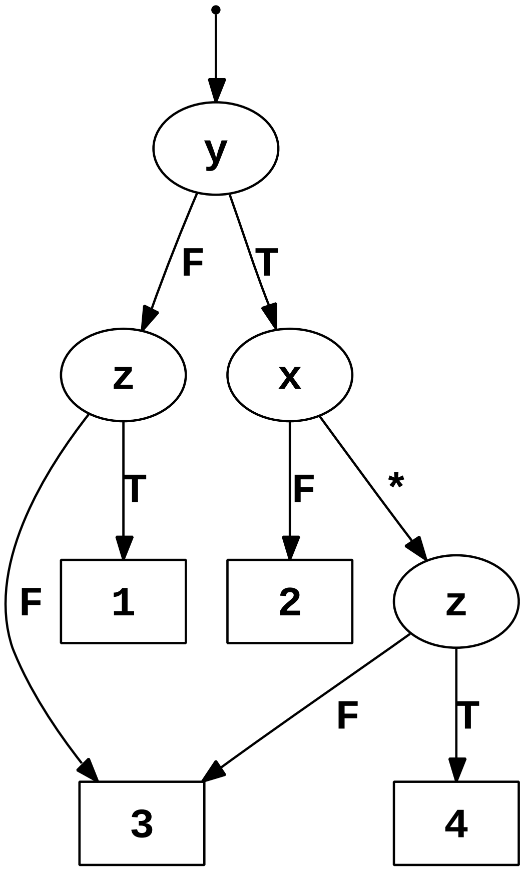


pR

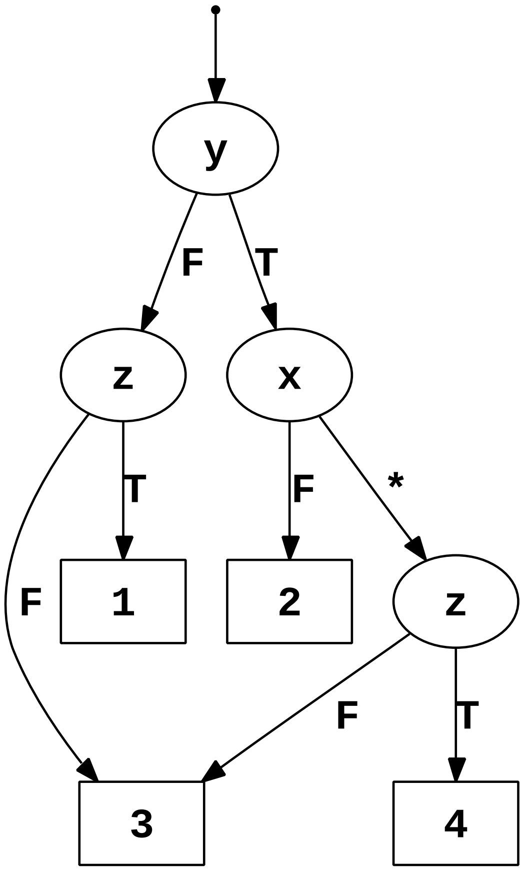


And also

nL



pL

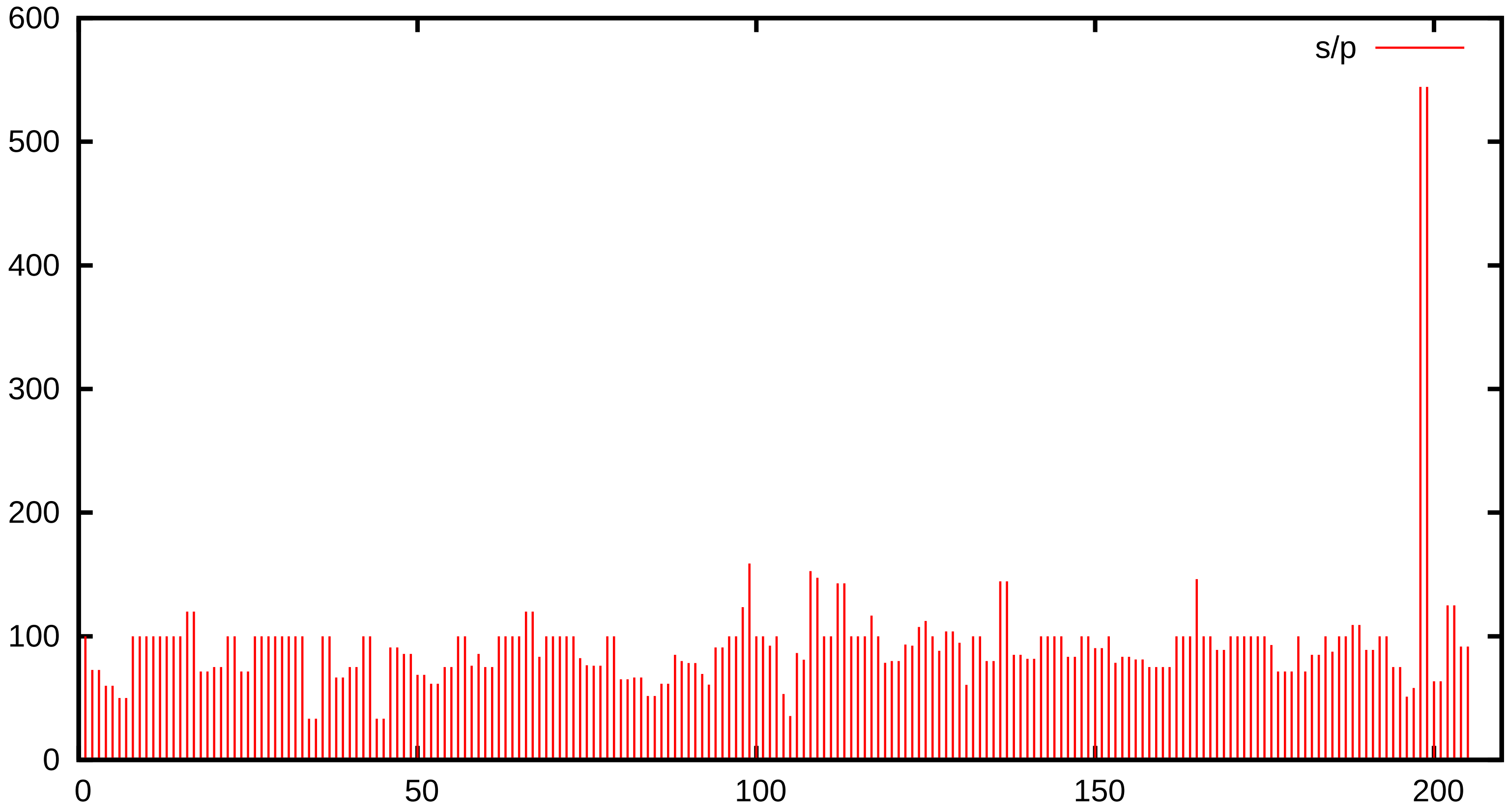


Experiments — Methodology

1. Select examples from a variety of real world programs (semi automatic selection of 103 matchings).
2. Apply all sequences of up to three heuristics (507 heuristics) to all examples, twice (ties left broken by L and R), with a prototype compiler.
3. Estimate decision tree quality by:
   (a) Number of test nodes in DAGs ($\sim$ code size).
   (b) Average path length ($\sim$ speed at runtime).
4. Now we have $2 \times 507 \times 2 \times 103$ numbers. . .

Dag size for **p**

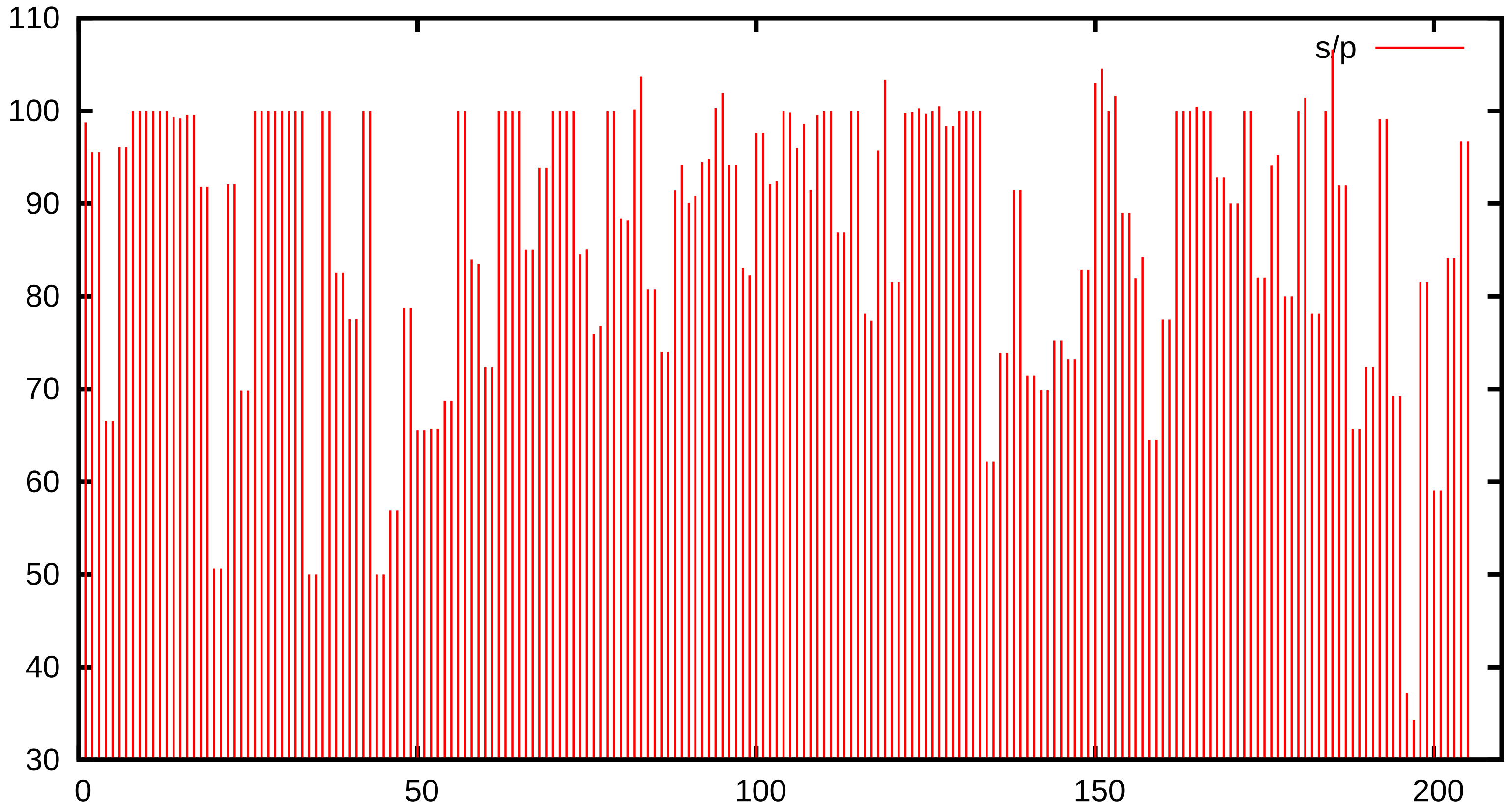
In fact for pL and pR.



Ratios w.r.t. OCaml match compiler (reference 100).

Average path length for p

In fact for pL and pR.



Ratios w.r.t. OCaml match compiler (reference 100).

Comparing

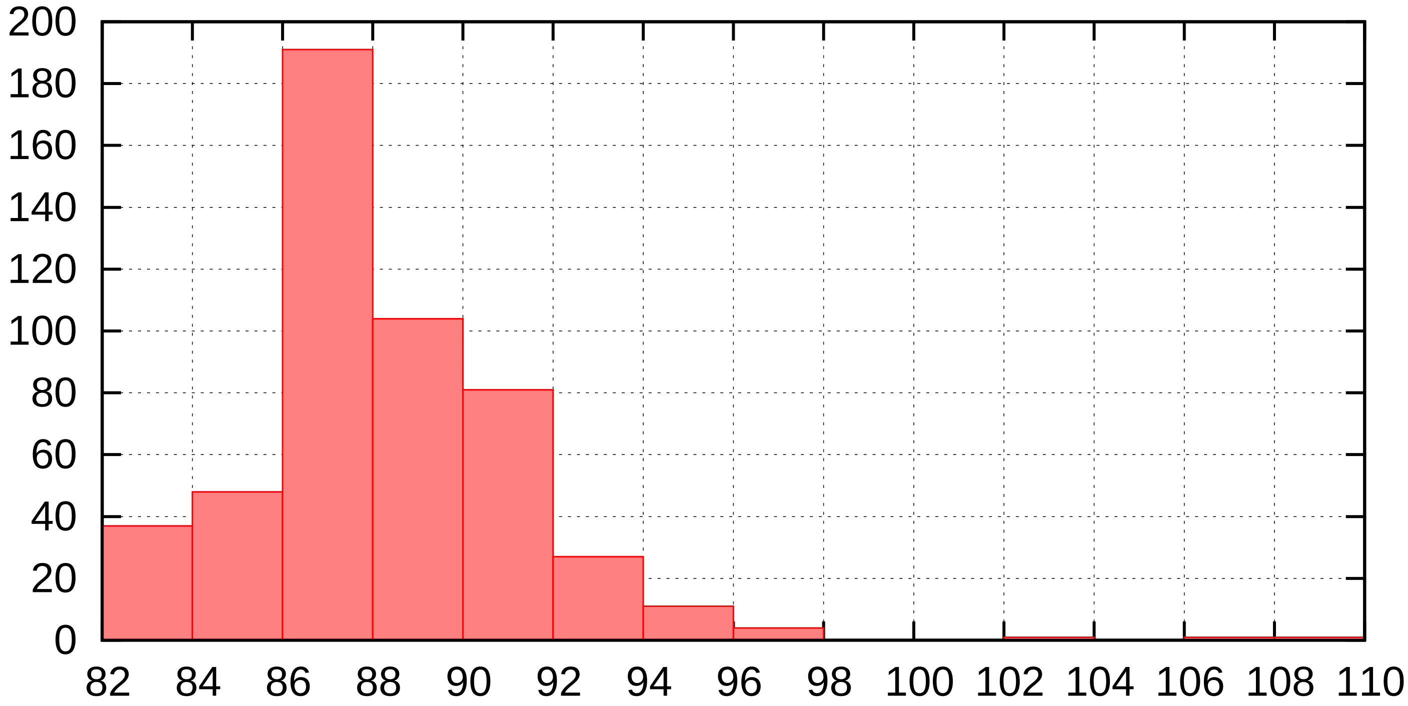
- Compute (geometric) means of data: yields 2 numbers per heuristic (size and average path length).

  For single heuristics:

	q	p	f	r	n	b	a	ℓ	d	N
Size	86	88	92	92	91	97	98	94	97	106
Path	86	86	87	89	86	94	91	87	88	92

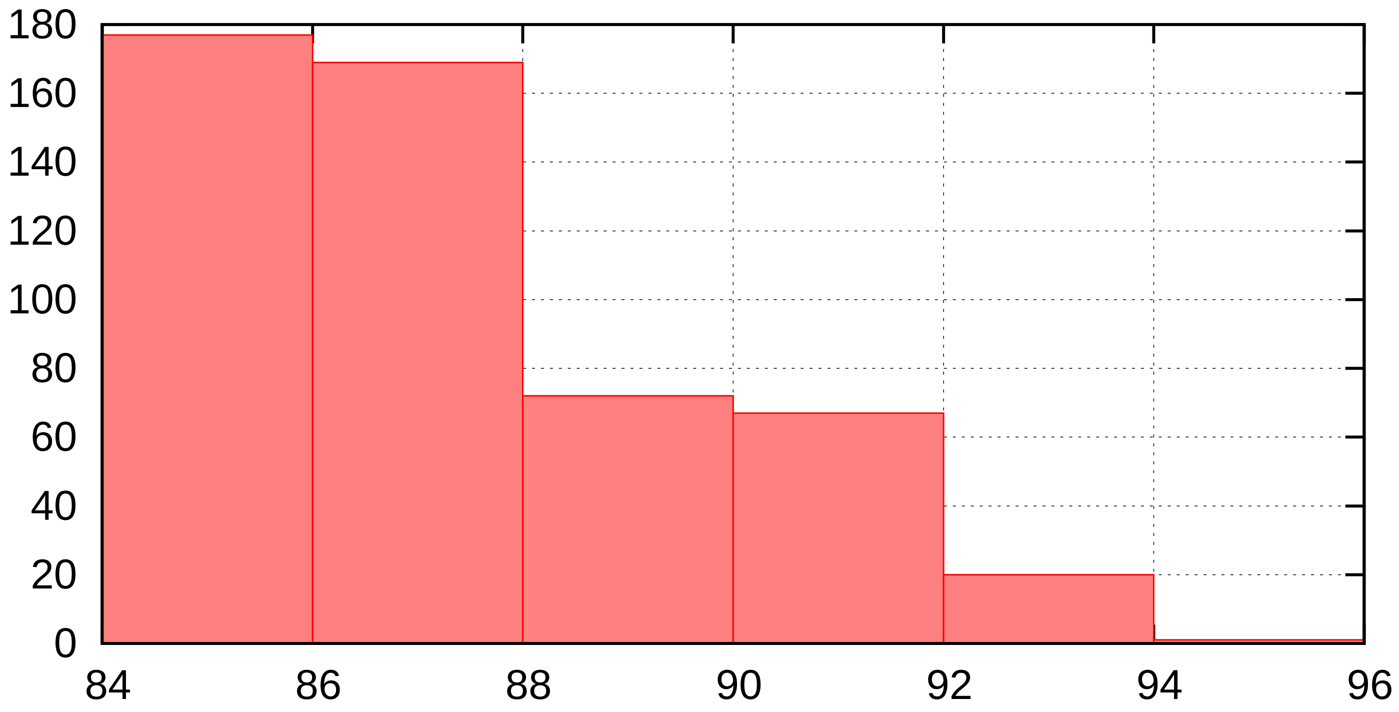
- But there are 507 heuristics: group them in classes.
- Find heuristics in the best classes for both size and path length.

Classes of heuristics, sizes



Best class: `qrp qrn qrd qr qdr qr`ℓ `pdb qra prb fbn pba pqb pbr pbd qnb qpb pb`ℓ `pbq fdb qrb qdb qbr qbp qbn qbd qb fb`ℓ `qba fr qb`ℓ `fra frd fr`ℓ `fbr fbd frn frb`

Classes of heuristics, path lengths



Best class: . . .

Best heuristics

Intersection of sizes in 82–84 and paths in 84–86:

`pba pbd pbl pbq pbr pdb pqb prb qb qba qbd qbl`
`qbn qbp qbr qdb qdr qnb qpb qr qra qrb qrd qrl`
`qrn qrp`

The winner is qb $(83.49/85.95)$ for instance, or pba $(83.75/85.83)$.

Or maybe fdb (SML/NJ, $83.51/86.07$).

Anyway, N is a looser $(106.38/92.47)$.

Actual performance

Implemented the new match compiler for OCaml.

Measured significant improvement (over standard OCaml) in final program speed for qba.

Conclusion

- Decision trees are competitive in practice, when optimized.
- Decision tress are easy to optimize. A simple algorithm + simple extensions:
  - A simple and effective heuristic (for instance qba).
  - Maximal sharing by hash-consing.