# Functional Programming:
# A Brief Introduction To Haskell

Henrik Nilsson

# What is Haskell?

- Designed in the late 80's by, among others, Prof. Paul Hudak, Yale.
- *Pure* Functional Language.
- *Non-strict* (or call-by-need) semantics.
- Statically typed, with parametric polymorphism (Hindley-Milner).

# Pure?

- *No* imperative side effects (in the traditional sense).
- Referential transparency.
- Equational reasoning.

Example:

```
x = f 1
y = x + x
```

is equivalent to

```
x = f 1
y = f 1 + f 1
```

## Non-strict semantics?

- Also
  - call-by-need, or
  - normal order reduction, or
  - outermost, leftmost reduction.
- Don't evaluate function arguments until needed.

The most common evaluation strategy is call-by-value, or applicative order reduction, or innermost first reduction, leading to *strict* semantics.

## Why non-strict semantics?

- Better termination properties than strict languages.
- In fact, best possible termination properties!
  (Church-Rosser Theorem II)
- "More declarative": can program without worrying too much about evaluation order.
- Useful programming techniques, e.g.:
  - circular programming,
  - "infinite" data structures.

## Drawbacks of normal-order reduction?

- Often inefficient, especially naive normal order reduction.
- Normal order reduction and imperative side effects *do not mix*!

Remedies?

- *Lazy evaluation* is a way to implement non-strict semantics that guarantees that an expression is evaluated *at most once*.
- *Actions* as first class values enjoying certain algebraic properties ensuring proper sequencing (a *Monad*).

# Some other functional languages

- Scheme
  - *Mostly* functional language.
  - *Strict* (or call-by-value) semantics.
  - Strongly, but not statically, typed.
- ML-Family: SML/NJ, Caml, OCaml
  - *Mostly* functional language.
  - *Strict* (or call-by-value) semantics.
  - Statically typed, with parametric polymorphism (Hindley-Milner).

# Aims

These two lectures aims at giving a brief overview of the following topics:

- Basic Haskell syntax; layout
- Parametric polymorphism and type inference
- Algebraic datatypes and pattern matching
- Recursion and recursive types
- Non-strict semantics (lazy evaluation) and its applications

# Circular programming

Given

```
data Tree = Leaf Int | Tree :^: Tree
```

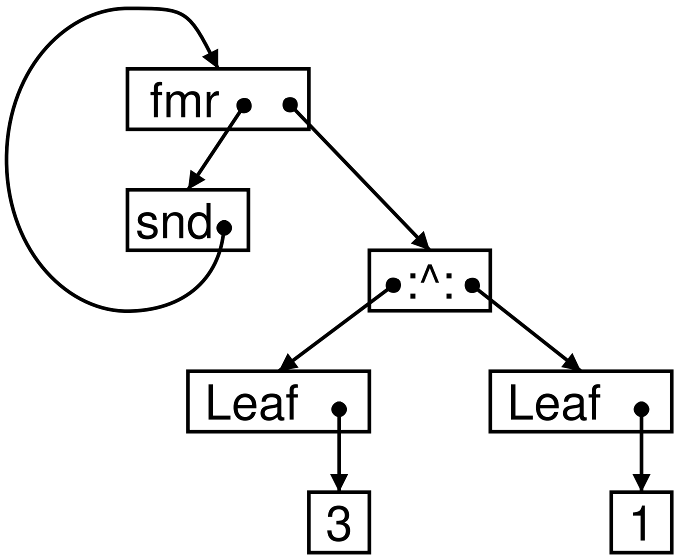
Suppose we would like to replace the leaf integers by the smallest integer in a tree. How many passes over the tree would be required?

# Circular programming (2)

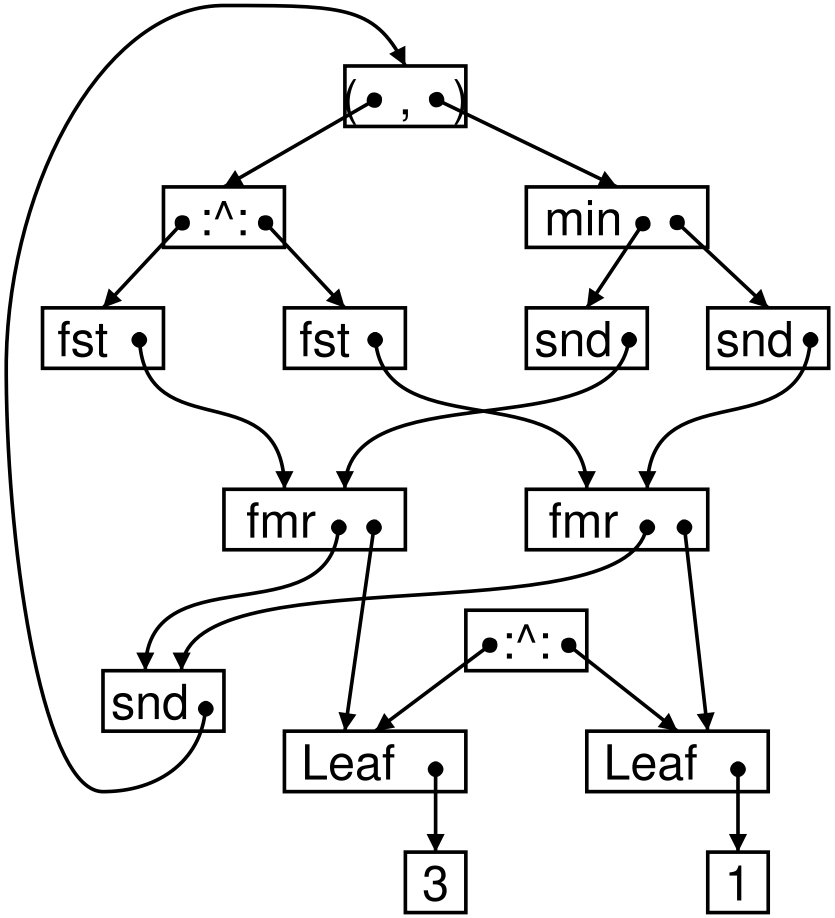
```
fmr :: Int -> Tree -> (Tree, Int)
fmr m (Leaf i) = (Leaf m, i)
fmr m (l :^: r) = (l' :^: r', min ml mr)
    where
        (l', ml) = fmr m l
        (r', mr) = fmr m r


replaceByMin :: Tree -> Tree
replaceByMin t = t'
    where
        (t', m) = fmr m t
```

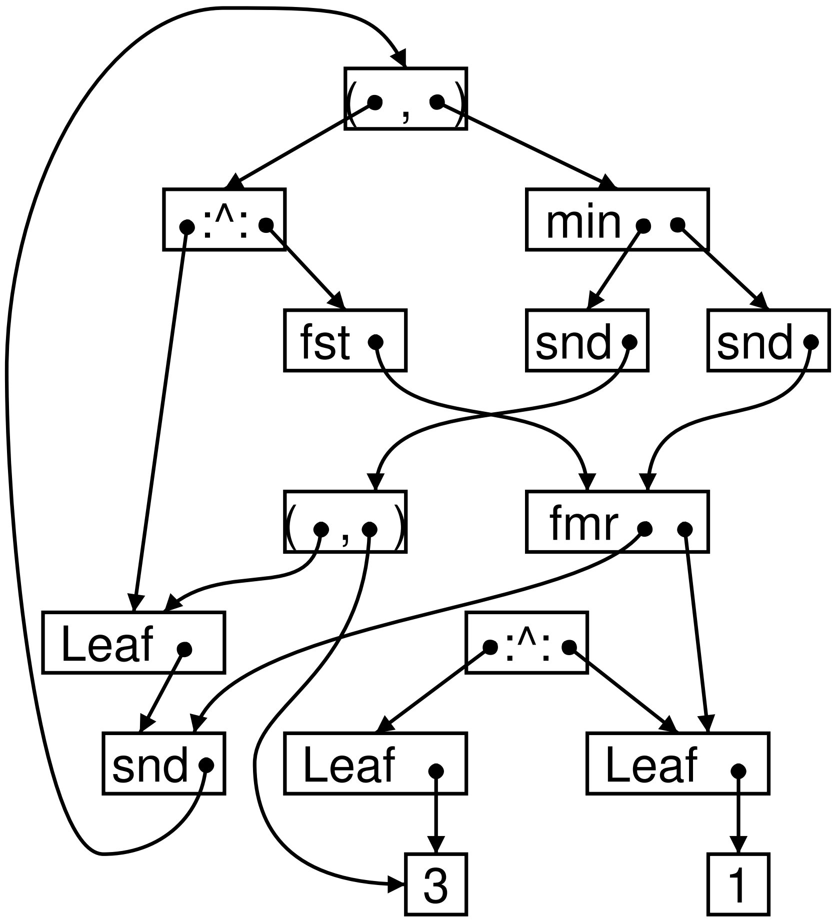
# Graph Rewriting (1)



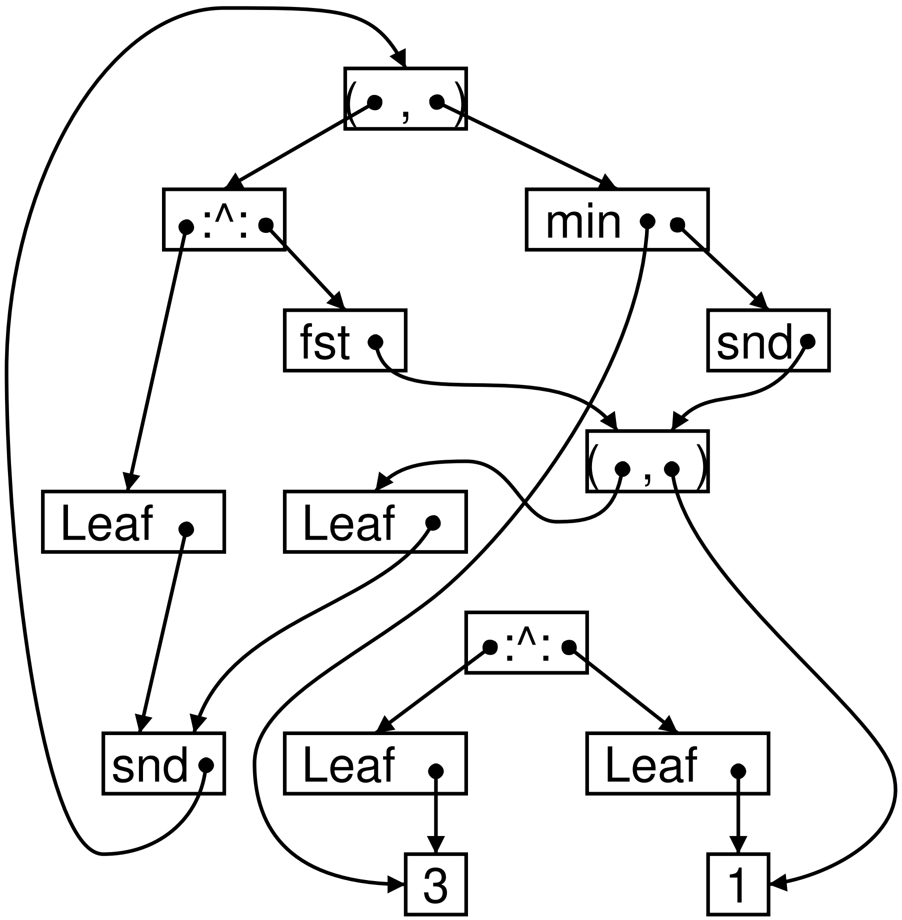
# Graph Rewriting (2)



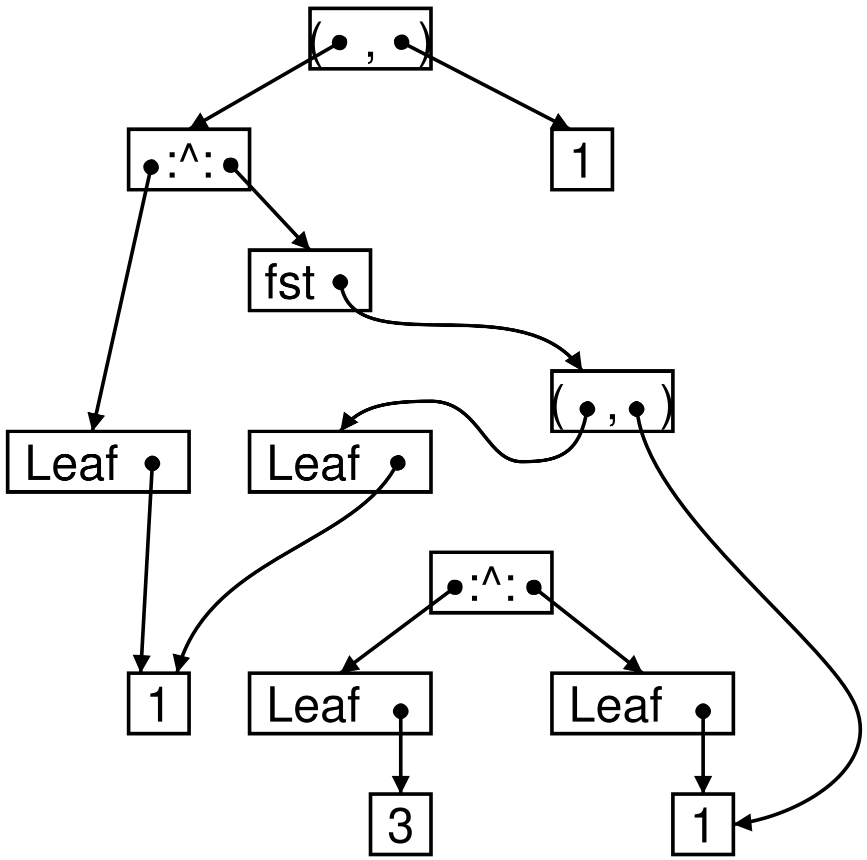
# Graph Rewriting (3)



# Graph Rewriting (4)



# Graph Rewriting (5)



# Graph Rewriting (6)