Lazy evaluation

1 Lazy Evaluation

- Not only is Haskell a pure functional language
- It is also evaluated lazily
- Hence, we can work with infinite data structures
- And defer computation until such time as it's strictly necessary

Definition: Lazy evaluation

Expressions are not evaluated when they are bound to variables. Instead, their evaluation is deferred until their result is needed by other computations

1.1 Evaluation strategies

- Haskell's basic method of computation is an application of functions to arguments
- Even here, though we already have some freedom

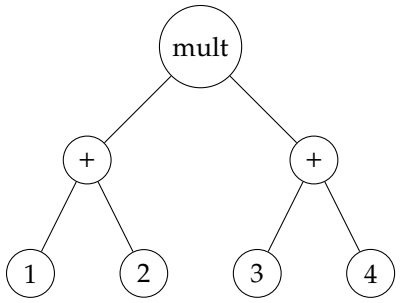
```
inc :: Int -> Int
inc n = n + 1
inc (2+3)
```

```
inc (2+3)
= inc 6 -- applying *
= 6 + 1 -- applying inc
= 7 -- applying *
```

```
inc (2+3)
= (2*3) + 1 -- applying inc
= 6 + 1 -- applying *
= 7 -- applying +
```

- As long as all expression evaluations terminate, the order we choose to do things doesn't matter
- We can represent a function call and its arguments as a graph
- Nodes in the graph are either terminal or compound. The latter are called reducible expressions or redexes

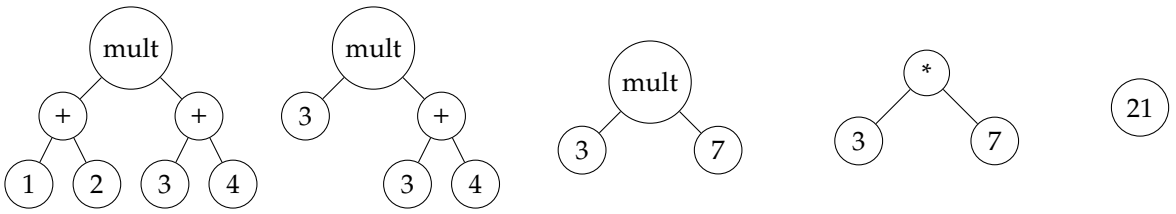
```
mult :: (Int, Int) -> Int
mult (x, y) = x*y
mult (1+2, 3+4)
```



- 1,2,3, and 4 are terminal (not reducible) expressions
- (+) and milt are reducible expressions

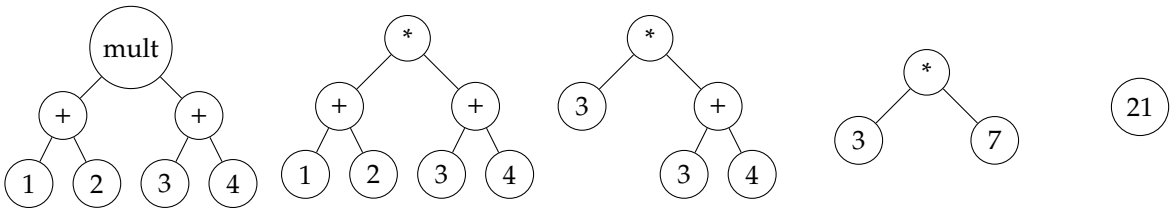
1.1.1 Innermost evaluation

- Evaluate "bottom up"
- First evaluate redexes that only contain terminal or irreducible expressions, then repeat
- Need to specify evaluation order at leaves. Typically : "left to right"



1.1.2 Outermost Evaluation

- Evaluate "top down"
- First evaluate redexes that are outermost, then repeat
- Again, need an evaluation order for children, typically choose "left to right"



1.2 Termination

- For finite expressions, both innermost and outermost evaluation terminate
- Not so for infinite expressions

1.3 Call by name or value

Call by value

- Also called eager evaluation
- Innermost evaluation
- Arguments to functions are always fully evaluated before the function is applied
- Each argument is evaluated exactly once
- Evaluation strategy for most imperative languages

Call by name

- Also called lazy evaluation
- Outermost evaluation
- Functions are applied before their arguments are evaluated
- Each argument may be evaluated more than once
- Evaluation strategy in haskell (and others)

1.4 Avoiding inefficiencies: sharing

- Straightforward implementation of call-by-name can lead to inefficiency in the number of times an argument is evaluated

```
square :: Int -> Int
square n = n * n
Prelude> square (1+2)
== (1 + 2) * (1 + 2) -- applying square
== 3 * (1 + 2) -- applying +
== 3 * 3 -- applying +
== 9
```

- To avoid this, Haskell implements sharing of arguments
- We can think of this as rewriting the evaluation tree into a graph

2 Controlling evaluation order

Definition: Normal form

The expression graph contains no redexes, is finite, and is acyclic
Data constructors are not reducible, so they "look like functions, there is no reduction rule

Definition: Weak head normal form (WHNF)

The expression graph is in normal form, or the topmost node in the expression graph is a constructor.
This allows for cycles

2.1 Evaluation rule

- Apply reduction rules (functions) outermost first
- Evaluate children "left to right"
- Stop when the expression graph is in WHNFF
- Function definitions introduce new reduction rules

2.2 Lazy evaluation in strict languages

- All (probably) languages have one place where they do something akin to lazy evaluation
- Boolean expressions to short circuit evaluation
- Avoids evaluating unnecessary expressions
- But not possible when assigning to variables
- Python generators are lazily evaluated
- Somewhat painful to work with when combining them

2.3 Strict functions

Definition: Strict function

A function which requires its arguments to be evaluated before being applied
Even when using outermost evaluation

Some functions in Haskell are strict (normally when working with numeric types)

2.3.1 Saving space

- Haskell uses lazy evaluation by default
- It also provides a mechanism for strict function application, using the operator ($!)

```haskell
($!) :: (a -> b) -> a -> b
f $! x -- evaluate x then apply f
```

- When using $!, the evaluation of the argument is forced until it is in weak head normal form

```
square $! (1 + 2)
== square $1 3 -- applying +
== square 3 -- applying $!
== 3*3 -- applying square
== 9 -- applying *
```

- This allows us to write functions that evaluate as if we had call-by-value semantics, rather than the default call-by-name
- Lazy evaluation can require a large amount of space to generate the expression graph
- In contrast, strict evaluation always evaluates the summation immediately, using constant space
- This kind of strict evaluation can be useful
- `sumwith` is ”just” a tail recursive left fold

```
sumwith  = foldl (+) 0
```

- For a strict version, which uses less space, we can use `foldl'`

```
import Data.Foldable
sumwith' = foldl' (+) 0
```

- This can have reasonable time saving for large expressions