CS 320: Concepts of Programming Languages

Wayne Snyder
Computer Science Department
Boston University

Lecture 02: Bare Bones Haskell

Syntax:
- Data == Abstract Syntax Trees
- Functions == Rewrite Rules on ASTs

Semantics:
- Evaluation == Rewriting
- Parameter Passing == Pattern-Matching

Review of Last Time….

- Programming Language = Syntax + Semantics
- Semantics is instantiated by another program (interpreter, compiler).
- Imperative languages (Java, C, ….) have statements that modify the state.
- State = Entire Memory
- Imperative program produces a sequence of state transitions.
- Imperative languages are hard to understand because tracing state transitions is hard!
- Functional programs remove (or control) the notion of state, using instead expressions which are rewritten by applying functions to subexpressions.
- Referential transparency = rewriting a subexpression ONLY changes that subexpression and there are no side-effects (no changes to state).

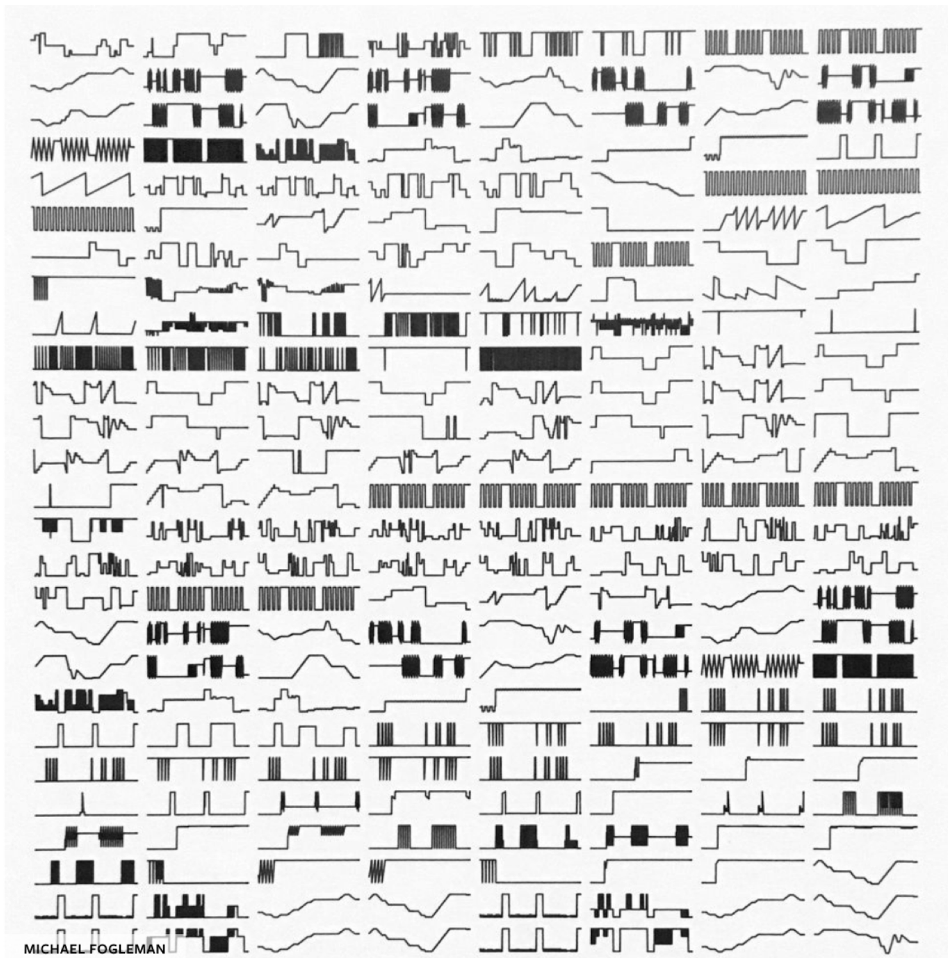
Review of Last Time….

POPULAR MECHANICS TECHNOLOGY CARS TOOLS DEFENSE STAY WARM!

Believe It or Not, This Is Pac-Man

These strange lines are a visualization of what happens inside the game.

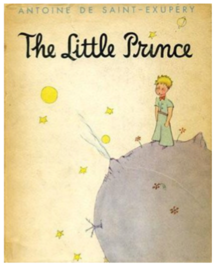
By Eric Limer Jan 25, 2019



Our Strategy for Learning FP through Haskell

- We are going to build a functional language (Haskell) from the "ground up," starting with the simplest possible "Turing complete" set of features (i.e., can do any computation), and adding features as we need them.
- These features will be "syntactic sugar" to make programming more convenient, and not fundamentally new ideas.
- We will maintain referential transparency, and when we introduce state, it will be as part of the expression.

"The true state of beauty exists not when there is nothing left to add, but when there is nothing left to take away." – Antoine de Saint-Exupery



Occam's Razor: "Entia non sunt multiplicanda praeter necessitatem."

"Less is more." – Ludwig Mies van der Rohe

Making Data in Bare-Bones Haskell

Recall: Programming language = Syntax + Semantics

Syntax = Data + Function Definitions

What is Data? Well, numbers, strings, lists, binary trees, hash tables, …..

Too complicated! Suppose all we have is the ability to say what syntax (words, basic punctuation) is data and what are functions….

How to create a piece of data?

Something

```
data Something
```

Making Data in Bare-Bones Haskell

What about creating a set of data objects? We need the data objects and we need a name (the "data type"):

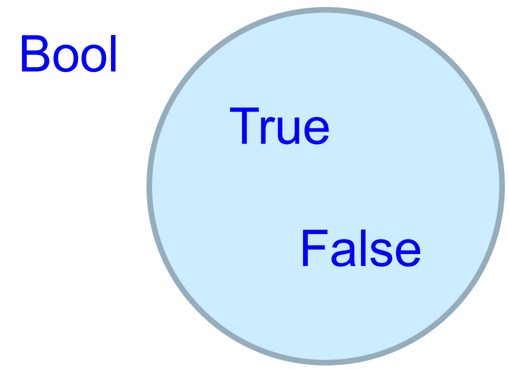
How about Booleans

"or"

```
data Bool = True | False
```

Name of data type

data objects in data type



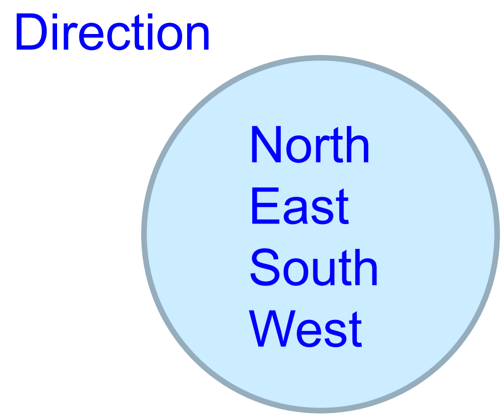
In Haskell, name of data objects and data types must be capitalized!

Data in Bare-Bones Haskell

More examples…..

data CS320Staff = Wayne | Mark | Cheng

data Direction = North | East | South | West



data ChessPieces = Pawn | Rook | Knight | Bishop | Queen | King

data Color = White | Black | Green | Blue | Red

Note: The actual names mean nothing!   Just syntax….

data A = B | C | D | E

Data in Bare-Bones Haskell

Structured data can be created by combining data declarations….

Simplest kind of structured data is a pair – two data objects combined together:

```
data BoolPair = Pair Bool Bool
```

Pair is called a Value Constructor because it constructs a data type from other data types.

We will sometimes just say "Constructor."

What do the actual structured data objects look like?

```
Pair True True          Pair True False

Pair False True         Pair False False
```

Data in Bare-Bones Haskell

```
data BoolPair = Pair Bool Bool
```

Parentheses can be used to clarify that this is a single, structured piece of data, but are not necessary:

```
Pair True True          Pair True False
Pair False True         Pair False False
```

Using parentheses: `(Pair True True)`

Value Constructor   Data types in the structure

NOTE: Incorrect syntax: Pair(True, False)

Data in Bare-Bones Haskell

We can create structured data from any (previously defined) data type:

data Direction = North | East | South | West

data Color = White | Black | Green | Blue | Red

data Arrow = Arrow Color Direction

Constructor — Data types in the structure

Note: It is allowed, and even encouraged, to use the same name for the name of the data type and the constructor.

Data objects of type Arrow:

`(Arrow Blue South)`   `Arrow  Green West`

But NOT: `Arrow  South Blue`   `Arrow Color Red`

Data in Bare-Bones Haskell

We can then add alternatives to create various kinds of structures for a single data type:

```
data  Direction = North | East | South | West
data Color = White | Black | Green | Blue | Red
data Arrow = Bare_Arrow
           | BlackArrow Direction
           | ColoredArrow Color Direction
```

Data objects of type Arrow:

```
(ColoredArrow Blue South)      Black_Arrow  West
     BareArrow
```

Data in Bare-Bones Haskell

Note that constructors take a particular sequence of data types, and (for now) ONLY those data types. You can't give multiple definitions of a constructor!

```
data  Direction = North | East | South | West
data Color = White | Black | Green | Blue | Red
data Arrow = BareArrow
           | Arrow Direction
           | Arrow Color Direction
```

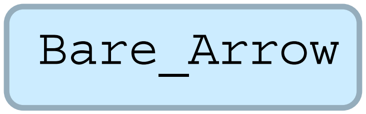
NOT ALLOWED!   Constructors must be unique!

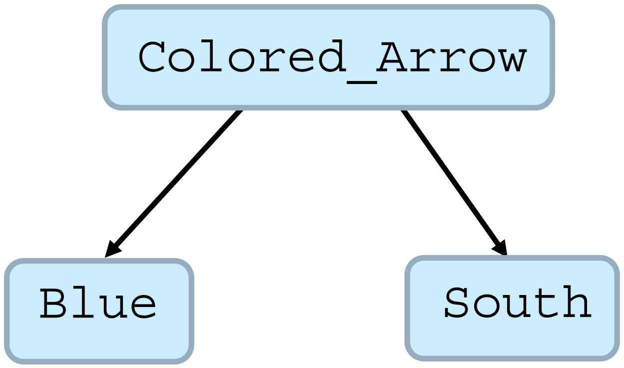
Data in Bare-Bones Haskell

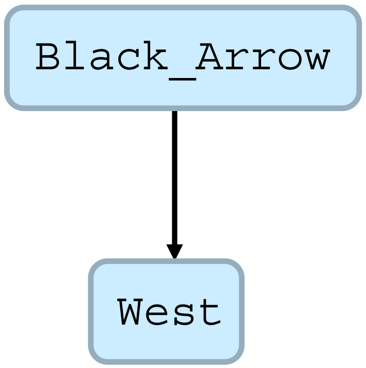
These data types have an obvious tree representation:

```
data Arrow = Bare_Arrow   | BlackArrow Direction
           | ColoredArrow Color Direction
```

```
Bare_Arrow
(ColoredArrow Blue South)
Black_Arrow  West
```







Data in Bare-Bones Haskell

We can also create recursive types, using the data type in its own declaration (see section 8.4 in Hutton):

data Nat = Zero | Succ Nat

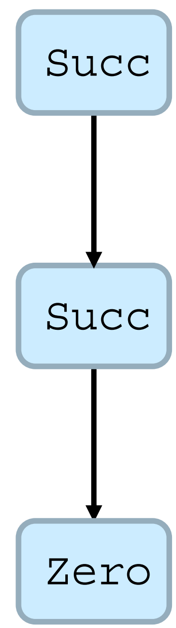
The constructor Succ takes a single data object of type Nat. This can be simple data object or structured (another Nat).

Data objects of type Nat:

```
Zero        (Succ Zero)

(Succ (Succ Zero))

(Succ (Succ (Succ (Succ Zero) ) ) )
```

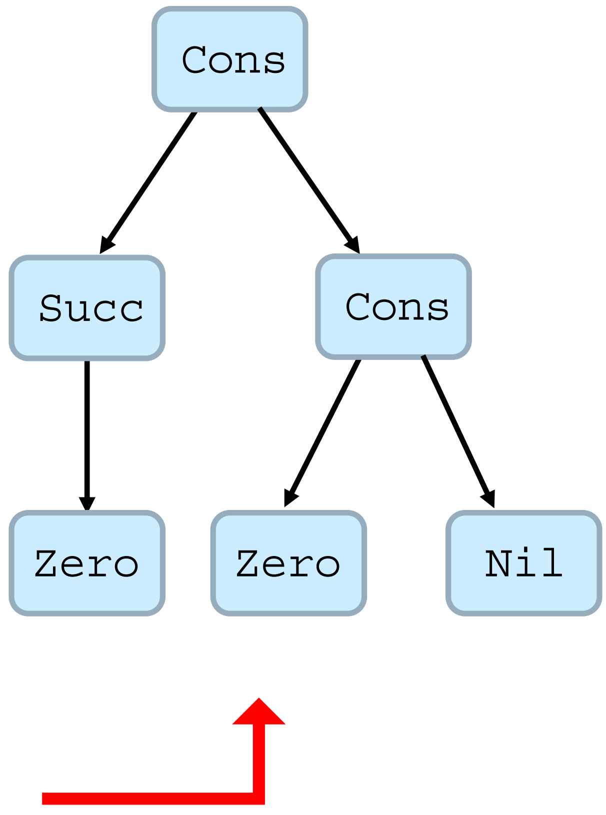


Data in Bare-Bones Haskell

How about Lists?

```
data Nat = Zero | Succ Nat

data List = Nil | Cons Nat List
```



Data objects of type List:

```
          Nil       Cons Zero Nil

(Cons (Succ Zero) (Cons Zero Nil) )
```

So, a Python list $[a_1,\ a_2,\ a_3,\ a_4,\ a_5]$ would be represented:

$(\mathtt{Cons}\ a_1\ (\mathtt{Cons}\ a_2\ (\mathtt{Cons}\ a_3\ (\mathtt{Cons}\ a_4\ (\mathtt{Cons}\ a_5\ \mathtt{Nil})\ )\ )\ )\ )$

Data in Bare-Bones Haskell

How about Binary Trees? (Hutton, p.97, adapted a bit!)

```
data Bool = True | False

data Tree = Leaf Bool | Node Tree Bool Tree
```

Data objects of type Tree:

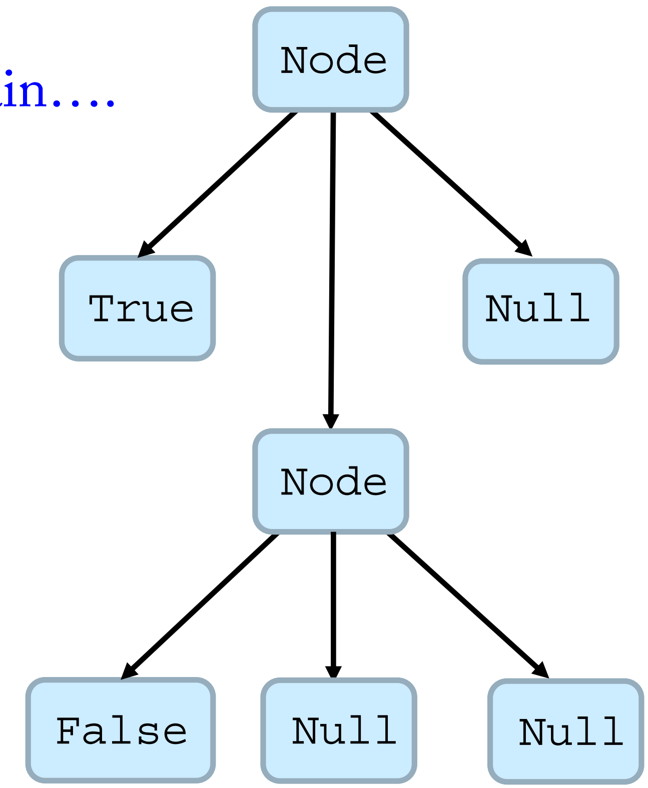
```
Leaf True     (Node (Leaf True) True (Leaf True) )

Node (Node (Leaf True) False (Leaf False))
     True
     (Leaf False)
```

```
NOT LEGAL:     Node False Leaf True Leaf False
```

Data in Bare-Bones Haskell

Hm… this doesn't allow for empty trees, so let's try again….




```
data Bool = True | False

data Tree = Null
          | Node Bool Tree Tree
```

Data objects of type the new type Tree:

```
Null                 (Node True Null Null)

Node True (Node False Null Null) Null
```

NOT LEGAL: `Node True Node False Null Null Null`

Functions in Bare-Bones Haskell

To define a function on the data objects, we give rules for rewriting a data object to another expression (possibly containing additional function calls).

```
data Bool = True | False
```

```
not False  =  True
not True   =  False
```

When we write an expression to the interpreter using a function name, it matches the function call to the rules:

```
> not False
True
> not True
False
```

Functions in Bare-Bones Haskell

```
data Bool = True | False
```

```
not False   =  True
not True    =  False
```

```
> not False
True
> not True
False
> not (not False)
False
```

Which is evaluated recursively:

```
not (not False)  => not True  => False
```

Functions in Bare-Bones Haskell

Evaluation of an expression by the interpreter proceeds as follows:

Scan the expression from the left (or: in post-order);
If a match between a sub-expression and the left-hand side of a rule is found, replace the subexpression by the right-hand side:

```
   not (not (not False) )
=> not (not True)
=> not False
=> True
```

Functions in Bare-Bones Haskell

Evaluation of an expression by the interpreter proceeds as follows:

Scan the expression from the left (or: in post-order);
If a match between a sub-expression and the left-hand side of a rule is found, replace the subexpression by the right-hand side:

```
   not (not (not False) )
=> not (not True)
=> not False
=> True
```

But function definitions without parameters are very limited!

So we have to add variables (= parameters).

Variables can be bound or "assigned" to any data object.

Functions as Rewrite Rules

To rewrite an expression, look for a rule which matches it – variables can match anything.

Rewrite, observing what bindings were made for variables.

Rules are tried in order!

```
data Bool = True | False
data Nat = Zero |  Succ Nat

not True = False

not False = True

cond  True   x  y   =   x      -- this is just
cond  False  x  y   =   y      -- an if-then-else


     (cond   False   Zero   (Succ Zero) )
         ↕
      not    True
```

no match!

Rule fails to match, try the next one!

Functions as Rewrite Rules

To rewrite an expression, look for a rule which matches it – variables can match anything.

Rewrite, observing what bindings were made for variables.

Rules are tried in order!

```
data Bool = True | False
data Nat = Zero |  Succ Nat

not True = False
not False = True

cond  True   x  y   =   x      -- this is just
cond  False  x  y   =   y      -- an if-then-else
```

```
(cond   False   Zero   (Succ Zero) )

 not    False
```

no match!

Rule fails to match, try the next one!

Functions as Rewrite Rules

To rewrite an expression, look for a rule which matches it – variables can match anything.

Rewrite, observing what bindings were made for variables.

Rules are tried in order!

```
data Bool = True | False
data Nat = Zero |  Succ Nat

not True = False

not False = True

cond  True   x  y   =   x      -- this is just
cond  False  x  y   =   y      -- an if-then-else
```

```
(cond   False   Zero   (Succ Zero) )

 cond   True    x      y
```

matches!

Functions as Rewrite Rules

To rewrite an expression, look for a rule which matches it – variables can match anything.

Rules are tried in order.

If a match is found, rewrite the expression, observing what bindings were made for variables.

```
data Bool = True | False
data Nat = Zero |  Succ Nat

not True = False

not False = True

cond  True   x  y   =   x      -- this is just
cond  False  x  y   =   y      -- an if-then-else
```

```
(cond   False   Zero   (Succ Zero) )

 cond   True    x       y
```

matches!   no match!

Rule fails to match, try the next one!

Functions as Rewrite Rules

To rewrite an expression, look for a rule which matches it – variables can match anything.

Rewrite, observing what bindings were made for variables.

Rules are tried in order!

```
data Bool = True | False
data Nat = Zero |  Succ Nat

not True = False

not False = True

cond  True   x  y   =   x      -- this is just
cond  False  x  y   =   y      -- an if-then-else


      (cond   False   Zero   (Succ Zero) )
        ↕
       cond   False   x       y
```

matches!

Functions as Rewrite Rules

To rewrite an expression, look for a rule which matches it – variables can match anything.

Rewrite, observing what bindings were made for variables.

Rules are tried in order!

```
data Bool = True | False
data Nat = Zero |  Succ Nat

not True = False

not False = True

cond  True   x  y   =   x      -- this is just
cond  False  x  y   =   y      -- an if-then-else
```

```
(cond   False   Zero   (Succ Zero) )
  ↕       ↕
 cond   False   x          y
```

matches!   matches!

Functions as Rewrite Rules

To rewrite an expression, look for a rule which matches it – variables can match anything.

Rewrite, observing what bindings were made for variables.

Rules are tried in order!

```
data Bool = True | False
data Nat = Zero |  Succ Nat

not True = False
not False = True

cond  True   x  y   =   x      -- this is just
cond  False  x  y   =   y      -- an if-then-else
```

```
(cond    False    Zero    (Succ Zero) )
  ↕        ↕        ↕
 cond    False      x         y
```

matches!   matches!   matches with x = Zero

Functions as Rewrite Rules

To rewrite an expression, look for a rule which matches it – variables can match anything.

Rewrite, observing what bindings were made for variables.

Rules are tried in order!

```
data Bool = True | False
data Nat = Zero |  Succ Nat

not True = False

not False = True

cond  True   x  y   =   x      -- this is just
cond  False  x  y   =   y      -- an if-then-else
```

```
(cond   False   Zero   (Succ Zero) )

 cond   False   x       y
```

matches! matches! matches with x = Zero matches with y = (Succ Zero)

Functions as Rewrite Rules

To rewrite an expression, look for a rule which matches it – variables can match anything.

Rewrite, observing what bindings were made for variables.

Rules are tried in order!

```
data Bool = True | False
data Nat = Zero |  Succ Nat

not True = False

not False = True

cond  True   x  y   =   x      -- this is just
cond  False  x  y   =   y      -- an if-then-else


     (cond   False   Zero   (Succ Zero) )

      cond   False    x          y

=>   (Succ Zero)     ( = y, where y = (Succ Zero))
```

x = Zero

y = (Succ Zero)

rewrites to

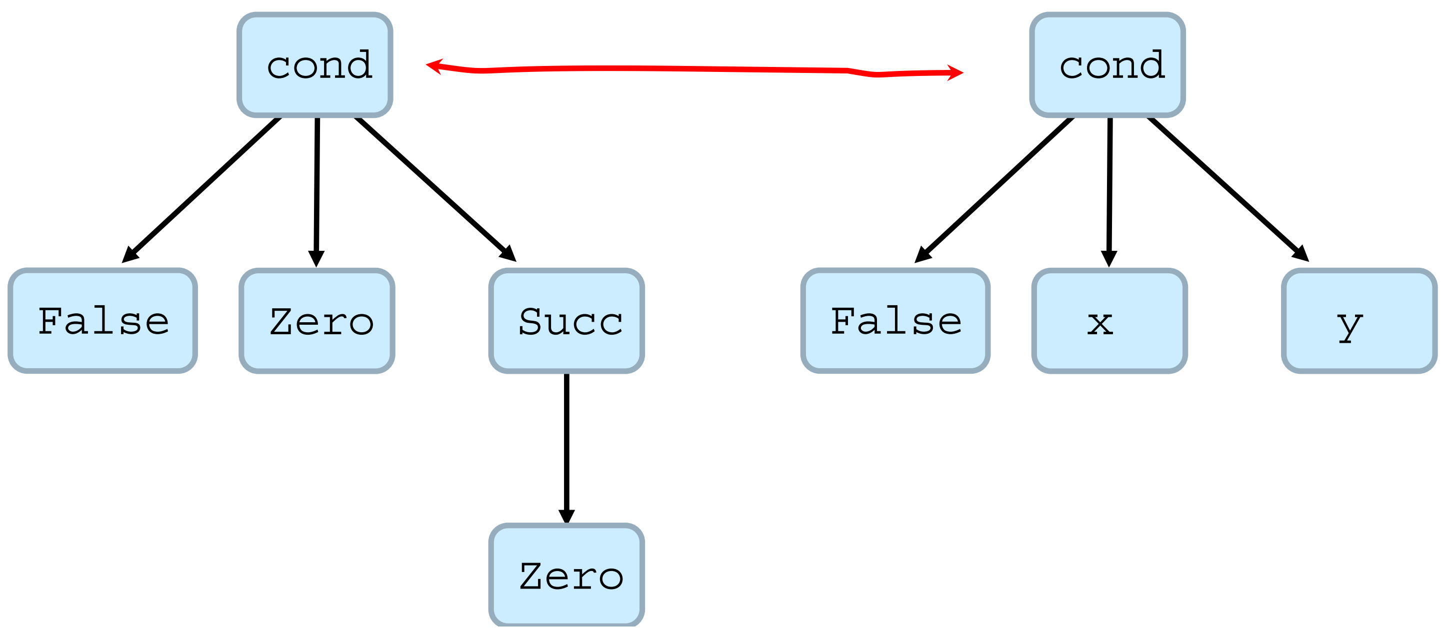
Functions as Rewrite Rules

```
data Bool = True | False
data Nat = Zero |  Succ Nat

cond  True   x  y   =   x
cond  False  x  y   =   y

     (cond   False   Zero    (Succ Zero) )
      cond   False    x          y
=>   (Succ Zero)    ( = y, where y = (Succ Zero))
```

A more precise version of this matching-and-rewriting model of computation is that we are rewriting trees, where function names and constructors label the nodes.... We traverse the trees preorder to determine matches....



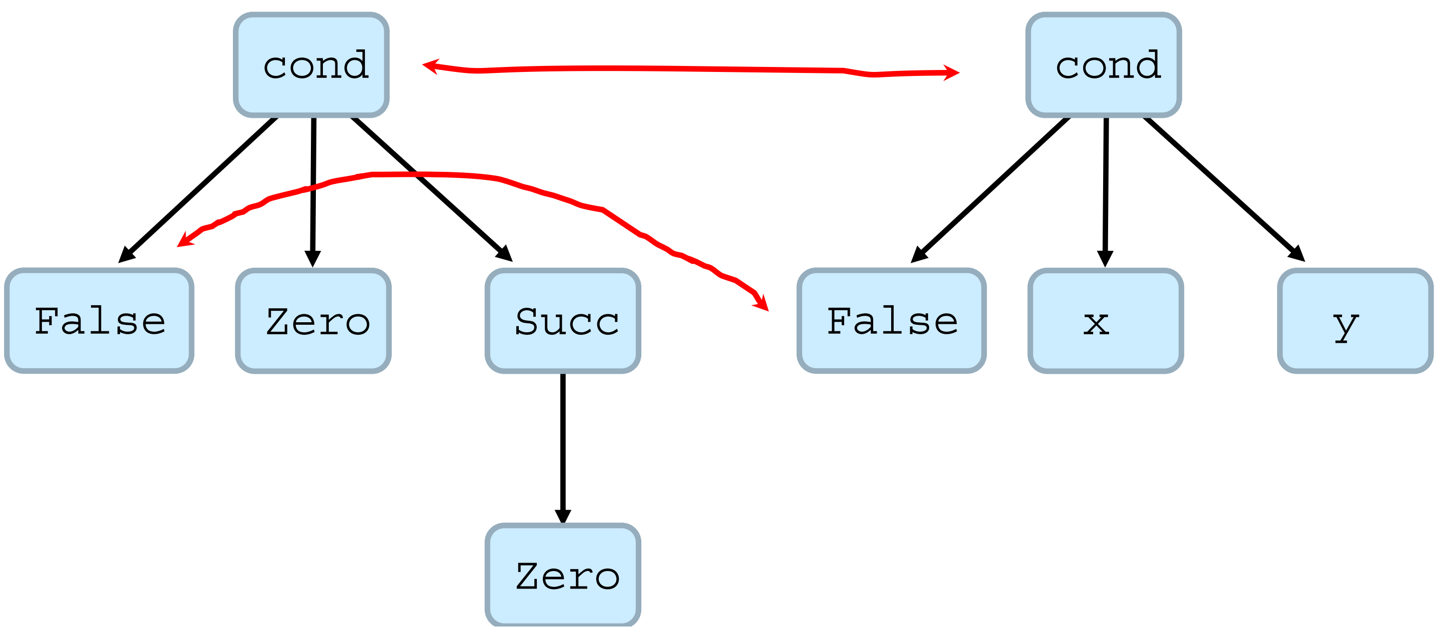
Functions as Rewrite Rules

```
data Bool = True | False
data Nat = Zero |  Succ Nat

cond  True   x  y   =   x
cond  False  x  y   =   y

     (cond   False   Zero   (Succ Zero) )
      cond   False    x          y
=>   (Succ Zero)     ( = y, where y = (Succ Zero))
```

A more precise version of this matching-and-rewriting model of computation is that we are rewriting trees, where function names and constructors label the nodes.... We traverse the trees preorder to determine matches....



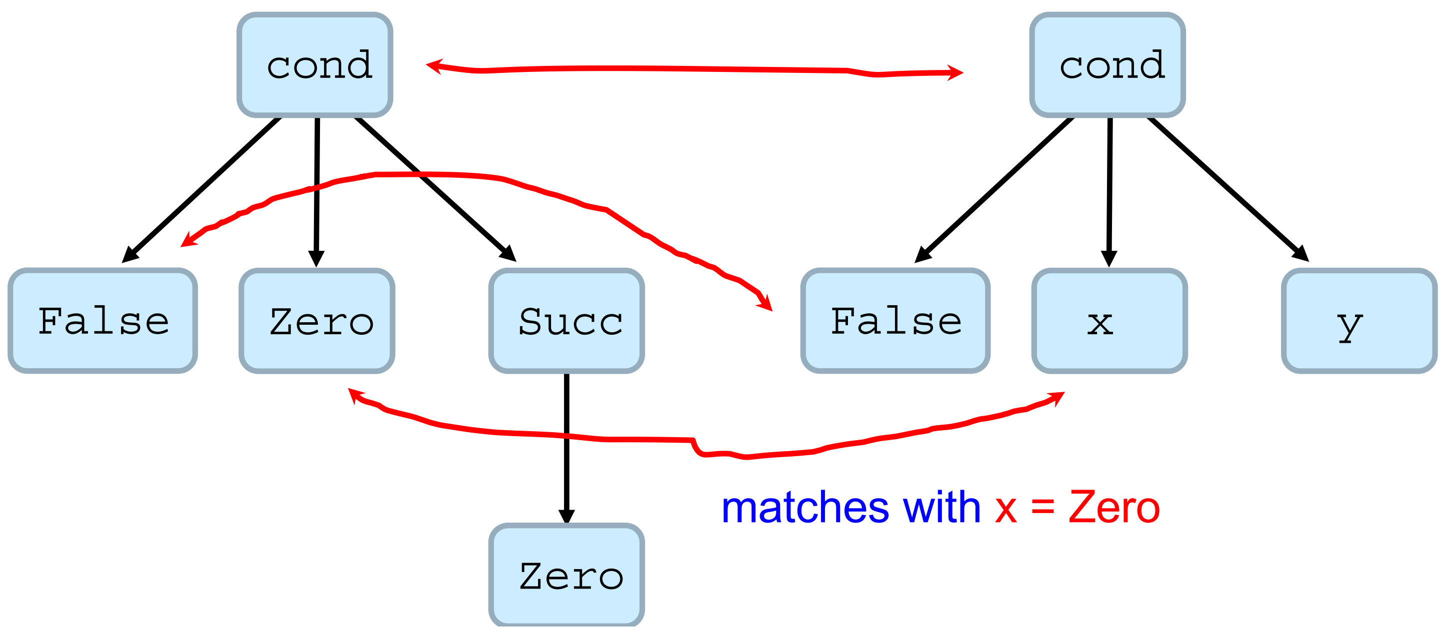
Functions as Rewrite Rules

```
data Bool = True | False
data Nat = Zero |  Succ Nat

cond  True   x  y   =   x
cond  False  x  y   =   y

      (cond   False   Zero    (Succ Zero) )
       cond   False    x          y
=>    (Succ Zero)     ( = y, where y = (Succ Zero))
```

A more precise version of this matching-and-rewriting model of computation is that we are rewriting trees, where function names and constructors label the nodes.... We traverse the trees preorder to determine matches....



Functions as Rewrite Rules

```
data Bool = True | False
data Nat = Zero |  Succ Nat

cond  True   x  y   =   x
cond  False  x  y   =   y

     (cond   False   Zero    (Succ Zero) )
      cond   False    x          y
=>   (Succ Zero)     ( = y, where y = (Succ Zero))
```

A more precise version of this matching-and-rewriting model of computation is that we are rewriting trees, where function names and constructors label the nodes.... We traverse the trees preorder to determine matches....