

Inductive Definitions

A bridge from math to computing

Review: two "programming paradigms"

Procedural/imperative

- A program executes a sequence of *steps*.
- Each step modifies *state*, i.e. variable values and memory content.
- examples: Python, Java, Javascript, Go, Swift, C, C++, Rust etc

Functional

- Evaluate *expressions*.
- Variables are names for expressions/values, not memory locations
- examples:
 - pure: **Haskell**
 - mostly functional: CAML, F#
 - imperative but has important functional part: Javascript, Python, Go, Swift, Rust, Scala

Aside on Haskell workflow

See the Ed note on this, and the accompanying Haskell file.

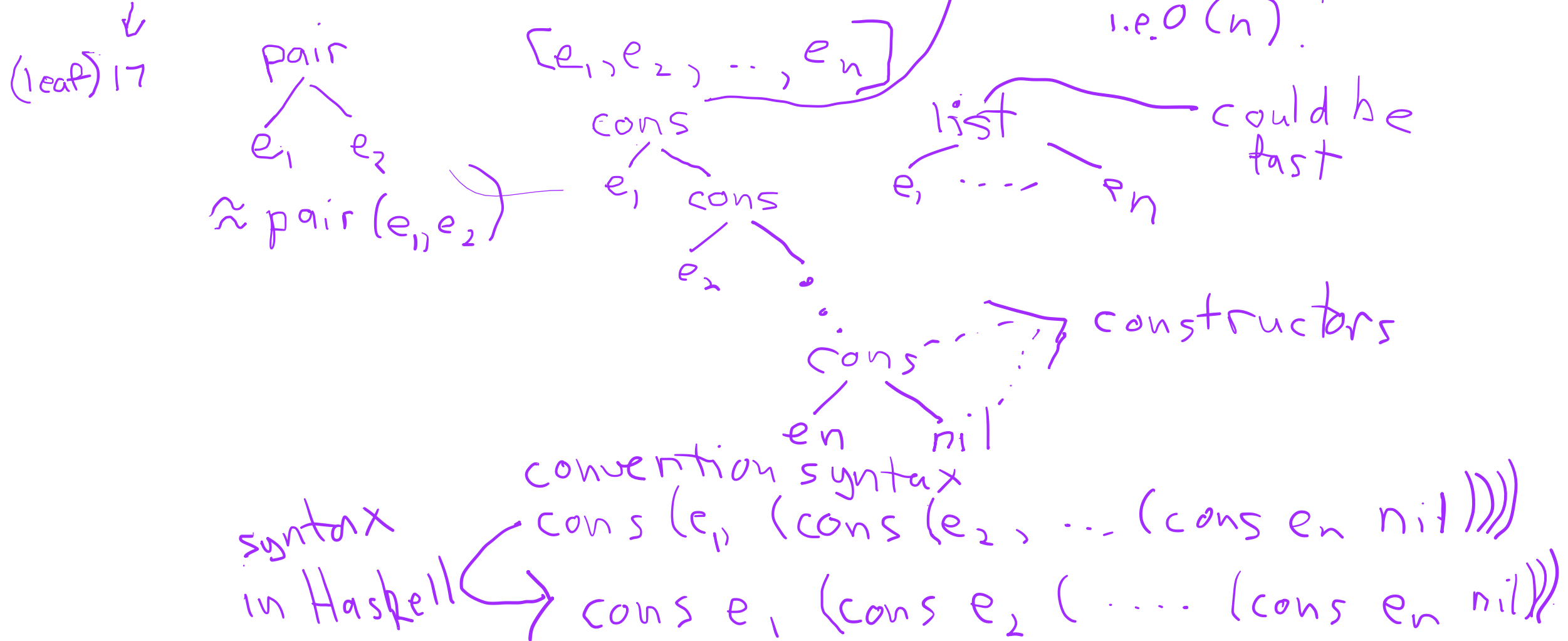
Recommended structure of Haskell file:

- start with type definitions
- function definitions with type declarations
- definitions giving names to test data
- optional definition of function `main`

Tree, tree, all is tree nodes

All data can be viewed as trees.

Constants, pairs, objects, lists (arrays), trees, databases.



Aside: in theory, only need one constant

Need one fixed constant to start with: ".".

Use it to build tree reps of natural numbers, integers, floats, strings.

Using cons, nil
 |
 pairing
 ↓
 constant } + == test
 if-then-else
 fst, snd
 functions w. recursive

How to define tree "datatypes": math approach.

How do we precisely specify a collection T of trees? Use an **inductive definition**.

1. Specify the "base" elements of T : objects that don't contain other elements of T (leaves).
2. Give rules for building elements (trees) of T from previously built elements of T (subtrees).

Compare to proofs by mathematical induction:

1. Base case: prove the property for the smallest elements of the set (e.g. 0 in the natural numbers).
2. For any building step (e.g. adding one), suppose the property is true for the parts, and show it's true for the built object..

A general definition of trees

An inductive definition defines a set of trees. What are trees?

1. A collection of nodes.
2. Each node has some other nodes as "children".
3. Each node has a "name" field. A name is just a string. Note that numbers can be represented as strings.

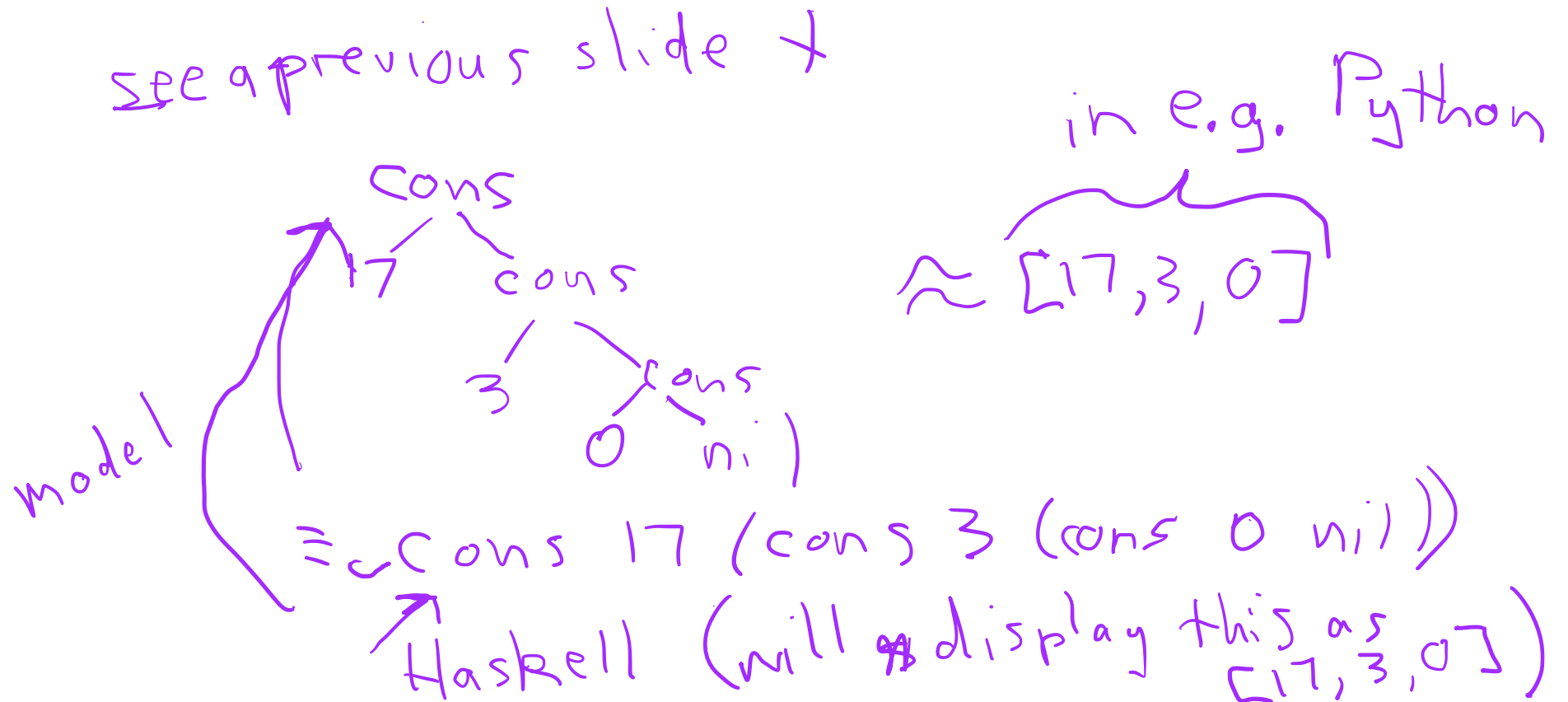
Of course, the structure needs to be "tree like", e.g. no cycles (such as node A parent of B which is parent of A).

Notation and example

It's best to think of trees pictorially, but those are hard to type in a text file.

Write $C\ t_1 \dots t_n$ for a tree with root node label C ("C" for constructor) and subtrees t_1, \dots, t_n .

Example:



A degenerate inductive definition

We can define the integers as an inductively defined set of trees Z :

1. Base case: for each integer n , the leaf node with name field n is in Z .
2. Construction (building) rules: none. Z only has "degenerate" trees, i.e. they're all leaves.

Inductive definition with rules: (integer) lists

Inductively define the set L of lists (more precisely, the set L of trees that we're calling lists) as follows.

1. Base case: the node with no children and with name field $[]$ is a list, i.e. $[] \in L$.
2. Construction (building) rule: if n is an integer (i.e. $n \in \mathbb{Z}$) and l is a list ($l \in L$), then the node with name field "::", and subtrees n and l , is a list, i.e. $(::)n l \in L$

\uparrow
cons

A tree is a list, according to the above definition, if and only if it is a base case or it can be derived from another list according to the rule in 2.

Build some lists

Something is a list (is in L), according to our inductive definition, exactly if it can be derived by the base case and the rules.

Using the base case, we immediately get $[] \in L$ is a list.

Since $[] \in L$ and $17 \in \mathbb{N}$,
 $:: 17 [] \in L$.

Since $:: 17 [] \in L$ and $3 \in \mathbb{N}$,
 $:: 3 (:: 17 []) \in L$.

Since $:: 3 (:: 17 []) \in L$ and $0 \in \mathbb{N}$,
 $:: 0 (:: 3 (:: 17 [])) \in L$. — prints as $[0, 3, 17]$

Notation for inductive definitions

The rules all have the form "if some objects have already been built then here's how to use those objects to build another object". We can write this using a horizontal line:

$$\frac{\text{assumed objects}}{\text{new object}}$$

Or, using an arrow:

$$\text{assumed objects} \Rightarrow \text{new object}$$

The assumptions above the line are called the *premises* of the rule. The part below the line is the *conclusion* of the rule.

Convenient fact: the base case can usually be viewed as a rule with no premises: the conclusion is true with no assumptions, i.e. it's always true.

List definition with new notation

$$\frac{}{[] \in L} (1) \qquad \frac{n \in Z \quad l \in L}{:: n \, l \in L} (2)$$

Or

$$\Rightarrow [] \in L \quad (1)$$

$$n \in Z, l \in L \Rightarrow :: n \, l \in L \quad (2)$$

Deriving lists more precisely

1. $[] \in L$	by rule 1.
2. $:: 17 [] \in L$	by rule 2 using line 1.
3. $:: 0 (:: 17 []) \in L$	by rule 2 using line 2.
4. $:: 3 (:: 0 (:: 17 [])) \in L$	by rule 2 using line 2.

Proof of



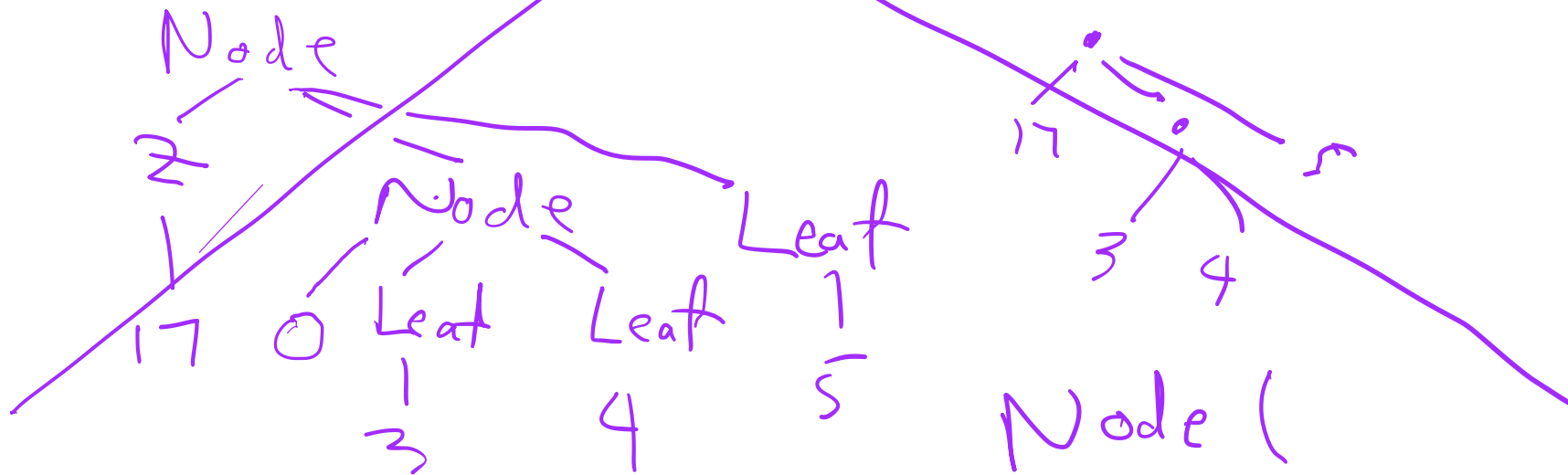
Example: binary trees

$\mathbb{Z}!$ n integer
 $\mathbb{Z} n$

Inductively define the set **BT** of binary trees with integer "info" as follows.

$n \in \mathbb{Z}$	$n \in \mathbb{Z}$	$t1 \in \text{BT}$	$t2 \in \text{BT}$
-----	-----	-----	-----
Leaf $n \in \text{BT}$	Node $n \ t1 \ t2 \in \text{BT}$		

Example tree with derivation and tree picture:



Computing with inductively defined objects

Let's focus on the BT example. There are two key observations.


1. A tree $t \in BT$ must be built using one of the rules, so one of the following **cases** is true.
 - a. t is *Leaf* n for some integer n ,
 - b. t is *Node* n $t1$ $t2$ for some number n and trees $t1$ and $t2$.
2. The trees $t1$ and $t2$ are **smaller** than t

We can compute using 1) pattern-matching case analysis and 2) recursion. E.g.

```
def incr(t):  
    if t is Leaf n:  
        Leaf(n+1)  
    else t is Node n t1 t2:  
        Node (n+1) (incr t1) (incr t2) # recursive calls on smaller trees
```

Running incr by hand

We can easily "run" it by just cranking through some equations.



```
incr(Node 3 (Node 4 (Leaf 5) (Leaf 6)) (Leaf 7))  
= Node (n+1) (incr(t1)) (incr(t2)) where n = 3, t1=(Node 4 ...), t2 = Leaf 7  
= Node (3+1) (incr(Node 4 (Leaf 5) (Leaf 6))) (incr(Leaf 7))  
= Node 4 (Node (4+1) (incr(Leaf 5)) (incr(Leaf 6))) (Leaf 8)  
= Node 4 (Node 5 (Leaf 6) (Leaf 7)) (Leaf 8) → final answer
```