Authors' Guide

Ryan Hota, Shubh Sharma, Arjun Maneesh Agarwal

Authors' Guide – Ryan Hota, Shubh Sharma, Arjun Maneesh Agarwal

This is (still!) an incomplete draft.
Please send any corrections, comments etc. to feedback_host@mailthing.com
Last updated May 25, 2025.

To a job well done

Table of Contents

Chapter Sketches

✅ Chapter 1: Function Definitions & Flow Control

- **Topics**: Function definition, pattern matching, recursion, induction, `let`, `where`, `if-then-else` as flow control.
- **Time**: 1 Class + 1 Tutorial
- **Author**: RSA
- **Notes**: No polymorphism; fully pen-and-paper before code.

—

🛠️ Chapter 2: Haskell Setup

- **Topics**: Minimal setup (hopefully using HaskellKISS) for different OS.
- **Time**: 1 Parallel Tutorial
- **Authors**:
  - **Windows**: R
  - **Linux**: S
  - **macOS**: A

—

🔤 Chapter 3: Basic Datatypes

- **Topics**:
  - `Bool`, `Int` vs `Integer` in extremely brief terms, `Char` (`ord`, `chr`)
  - Use of `.` vs `"."`
- **Time**: 1 Class + 1 Tutorial
- **Author**: A
- **Notes**: No polymorphism yet.
- **Assignment**: Number Theory & Logic Ops

—

🧮 Chapter 4: Types as Sets

- **Topics**:
  - Tuples as $\times$, `Either` as $\cup$, `(->)` as A^B, `::` as $\in$
  - Currying (concept only), implicit parentheses
  - Basic set theory concepts
- **Time**: 1 Class + 1 Tutorial
- **Author**: R

—

📃 Chapter 5: Lists

- **Topics**:
  - List definition & comprehension
  - Lists as syntax trees
  - Operations: `head`, `tail`, `!!`, `elem`, `drop`, `take`, `splitAt`
  - Merge sort, infinite lists
  - Code Examples:

```
l = 0 : l
l n = n : l (n+1)
l a b = a : l b (a+b)
```

- **Time**: 2 Classes + 2 Tutorials
- **Author**: R

—

🔁 Chapter 6: Polymorphism & Higher Order Functions

- **Topics**:
  - Intro to polymorphism
  - Higher-order functions
  - Operators and functions:

    ```
    ($) :: (a → b) → a → b
    a → b → (a, b)
    curry, uncurry
    ```
- **Time**: 1 Class + 2 Tutorials
- **Author**: S

—

🔃 Chapter 7: Advanced List Operations

- **Topics**:
  - `map`, `filter`, Cartesian product, first through list comprehension, then explicitly defined
  - Quick sort through list comprehension
  - `zip`, `zipWith`
  - Folds, scans (with syntax tree understanding)
  - Miller–Rabin primality test
- **Time**: 2 Classes + 3 Tutorials
- **Author**: A

—

📊 Chapter 8: Precomp Data Structures

- **Topics**:
  - Define recursion in recursive data types
  - Define whatever happened in the basic datatypes section.
  - Define Nat, List, Tree
- **Time**:
- **Author**: A

🧵 Chapter 9: Computation as Reduction

- **Topics**:
  - Reduction-based computation (skip Big O)
  - Syntax trees, lazy evaluation
  - Examples:
    - Fibonacci via infinite list
    - Test if the following works:

      ```
      (map recip [-5..]) !! n
      ```
- **Time**: 1 Class + 2 Tutorials
- **Author**: S

☣️ Chapter 10: Complexity

- **Topics**:
  - Some Notion of complexity that is pretty theoretical
- **Time**:
- **Author**: A

📊 Chapter 11: Post Comp Data Types

- **Topics**:
  - Queue
  - Segment Trees
  - BST
  - Set
  - Map
- **Time**:
- **Author**: A

🏫 Chapter 12: Typeclasses

- **Topics**:
  - Recall Polymorphism
  - `deriving`
  - Under the hood of `deriving`
  - Custom Classes
- **Author**: R
- **Time**:

⏩ Chapter 13: Monads

- **Topics**:
  - Functors
  - Applicative Functors?
  - Monads:
    - Theory
    - Do notation
    - Simple Writer - Cost as `(,) Integer`
    - Maybe Monad
    - Simple Reader - `(->) x`
    - Simple State - Light Switch
    - Monoid Monad
    - Simple Writer - IO
- **Author**: R
- **Time**: 5 Classes + 5 Tutorials

—

🗂️ Directory Structure

In general, do not make new typst files in the chapter area, just edit the previously existent ones. If you use a figure, please save it as a seperate .typ or .asy file in figures and import when required.

```
haskell-course/
├── Modules
│   ├── Book.typ
│   ├── Box.typ
│   ├── Chapter.typ
│   ├── Code.typ
│   ├── Contents.typ
│   ├── Definition.typ
│   ├── Exercise.typ
│   ├── Prelude.typ
│   ├── Proof.typ
│   ├── Tree.typ
│   ├── style.typ
│   ├── theme.tmTheme
│   └── theorems.typ
├── chapters-typ
│   ├── ch01_functions.typ
│   ├── ch02_setup_linux.typ
│   ├── ch02_setup_mac.typ
│   ├── ch02_setup_win.typ
│   ├── ...
│   ├── ch13_monad.typ
│   └── example_chapter.typ
├── example.pdf
├── extra-typ
│   ├── appendix.typ
│   └── preface.typ
├── figures
├── generate.sh
├── guide.md
├── guide.pdf
├── guide.typ
├── licenses
│   ├── code_license.txt
│   └── text_license.txt
├── main.pdf
├── main.typ
└── tutorials-typ
```

Tutorials, assignments and solutions refer to tutorial handouts(if needed), class and tutorial assignments and their solution files. Keep them as .typ/.tex files for now, the required .hs/.lhs files will be generated later.

Also, if you need to cite something, cite it at the end of your chapter as a comment starting with cite.

```
// cite:
// citation 1
// citation 2
```

I will at some point make a script to compile citations. OR We can use https://typst.app/docs/reference/model/cite/ and Hayagriva

Pedagogy

Personality of Narrator

We always use the "we" grammatical turn as far as possible, and a few steps further still.

How to know whether a Concept has been Learnt

By testing whether they can parse Haskell in natural language

We can test a student upon their knowledge of a Haskell function by asking the student to narrate in detail in a natural language such as English, the steps the function is taking in the execution of its definition.

For example,

Consider the following problem : We have to make a function that provides feedback on a quiz. We are given the marks obtained by a student in the quiz marked out of 10 total marks. If the marks obtained are less than 3, return `'F'`, otherwise return the marks as a percentage -

λ function to an either type

```haskell
feedback :: Integer -> Either Char Integer
--                     Left ~ Char,Integer ~ Right
feedback n
  | n < 3     = Left  'F'
  | otherwise = Right ( 10 * n ) -- multiply by 10 to get percentage
```

You then ask the student to describe in detail how the function works.

They should ideally answer - "

Let `feedback` be a function that takes an `Integer` as input and returns `Either` a `Char` or an `Integer`.

As `Char` and `Integer` occurs on the left and right of each other in the expression `Either Char Integer`, thus `Char` and `Integer` will henceforth be referred to as `Left` and `Right` respectively.

Let the input to the function `feedback` be `n`.

If `n<3`, then we return `'F'`. To denote that `'F'` is a `Char`, we will tag `'F'` as `Left`. (remember that `Left` refers to `Char`!)

`otherwise`, we will multiply `n` by `10` to get the percentage out of 100 (as the actual quiz is marked out of 10). To denote that the output `10*n` is an `Integer`, we will tag it with the word `Right`. (remember that `Right` refers to `Integer`!)

"

What does it Mean to Teach a Haskell Concept ?

Teaching is a very abstract notion. We can make that notion more explicit by assuming that what we actually want to do is get the student to a position where they can pass the test described in the above section with flying colours, i.e.,

$$\text{Teaching} \approx \text{Teaching to pass the test of the previous section}$$

How to Teach a Haskell Concept

Assuming the suppositions of the previous sections hold any water, the method to teach a Haskell concept appears quite simple -

$$\text{Show the students how to parse Haskell in natural language over and over again until they get it}$$

Using Imported Functions

Code Block

We can write any text that is meant to be code using the usual syntax of using backticks (`).

Inline Code Block

We can make a code block in the middle of a line.

Source	Rendered
Let us take the \`hello\` function as an example.	Let us take the `hello` function as an example.

Floating Code Block

We can make a code block that floats out-of-line on its own.

````
```
hello :: any → String
hello      _  =  "world"
```
````

```
hello :: any → String
hello      _  =  "world"
```

Literate Haskell

We are using `markdown-unlit` as our literate haskell pre-processor. It only processes those ***floating*** code blocks at ***zero indentation depth*** which have been ***marked as haskell*** code blocks. So, if you want your code to be visible to lhs, you need to write "haskell" immediately after the 3 backticks, leaving no space in between.

````
```haskell
hello :: any → String
hello      _  =  "world"
```
````

```haskell
hello :: any → String
hello      _  =  "world"
```

Hiding a Code Block

We can hide a code block by putting it inside the Typst function `#metadata()`.

````
#metadata(
```
hello :: any → String
hello      _  =  "world"
```
)
````

This can be ***useful*** if paired with the syntax required for ***Literate Haskell***, as then we can have code that doesn't appear on the PDF, but still executable by Literate Haskell.
And thus readers would stilll be able to access it in GHCi without it taking up valuable reading space on the PDF.

Code Block Title

We will often find it a good idea to title a code block, because then it will show up in the table of contents, in the glossary and can be referenced.

If the first line of the code in a ***floating*** code block begins with `-- |`, then the rest of that line will be taken as the title.

````
```
-- | helloWorld function
hello :: any → String
hello      _  =  "world"
```
````

λ **helloWorld function**

```
hello :: any → String
hello      _  =  "world"
```

Referencing a Code Block

A ***titled*** definition with a ***unique title*** can referenced by the usual syntax.

Source	Rendered
Recalling @code_of_helloWorld_function, we can proceed.	Recalling λ **helloWorld function**, we can proceed.

Definition

To use this module, we need to `#import "../Modules/Definition.typ": def`

Definition Box

We can call the function `#def()` upon some `content` put that `content` in a ***floating*** definition box. We can put any text that is meant to be definition in a definition box.

```
#def[
  The empty set is the set that
contains no elements or equivalently,
${}$.
]
```

The empty set is the set that contains no elements or equivalently, $\{\}$.

Emphasizing the Subject

To increase readability, we can emphasize the subject of the dinition by wrapping it in `**`.

```
#def[
  The *empty set* is the set that
contains no elements or equivalently,
${}$.
]
```

The **empty set** is the set that contains no elements or equivalently, $\{\}$.

Emphasizing the Definition

To increase readabilty, we can emphasize ***the part of the text that is the actual definition*** by wrapping that part in `__`.

```
#def[
  The empty set is the _set that
contains no elements_ or equivalently,
_${}$_.
]
```

The empty set is the ***set that contains no elements*** or equivalently, $\{\}$.

Definition Box Title

We will often find it a good idea to title a definition, because then it will show up in the table of contents, in the glossary and can be referenced.
We can set the `subject` settable argument of the `#def()` function to a `string` if we want to add a title.

```
#def(subject: "empty set")[
  The empty set is the set that
contains no elements or equivalently,
${}$.
]
```

÷ **empty set**

The empty set is the set that contains no elements or equivalently, $\{\}$.

Referencing a Definition

A ***titled*** definition with a ***unique title*** can referenced by the usual syntax.

```
Recalling @definition_of_empty_set, we
can proceed.
```

Recalling ÷ **empty set**, we can proceed.

Exercise

To use this module, we need to `#import "../Modules/Exercise.typ": exercise`

Exercise Box

We can call the function `#exercise()` upon some `content` put that `content` in a ***floating*** exercise box.
We can put any text that is meant to be an exercise in an exercise box.

```
#exercise[
  If a type `T` has $n$ elements, then
how many elements does `Maybe T` have?
]
```

If a type `T` has n elements, then how many elements does `Maybe T` have?

Exercise Box Title

We will often find it a good idea to title an exercise, because then it will show up in the collection exercises when we use `#exercises`, in the glossary and can be referenced.
We can set the `subject` settable argument of the `#exercise()` function to a `string` if we want to add a title.

```
#exercise( subject : "maybe" )[
  If a type `T` has $n$ elements, then
how many elements does `Maybe T` have?
]
```

X **maybe**

If a type `T` has n elements, then how many elements does `Maybe T` have?

Referencing an Exercise

A ***titled*** exercise with a ***unique title*** can referenced by the usual syntax.

```
Recalling @exercise_of_maybe, we can
proceed.
```

Recalling X **maybe**, we can proceed.

Proof

To use this module, we need to `#import "../Modules/Proof.typ": proof`

Proof Environment

We can call the function `#proof()` upon some `content` to prepend that `content` with a "Proof" tag. We can treat any text that is meant to be an proof in this manner.

```
#proof[
  Here is a proof.
]
```

Proof Here is a proof.

Theorem Environment

To add line with a "Theroem" tag above the proof, we can set the `thm` settable argument of the `#proof` function to that line.

```
#proof( thm: [This is a theorem
statement.] )[
  Here is a proof.
]
```

Theorem This is a theorem statement.

Proof Here is a proof.

Tree

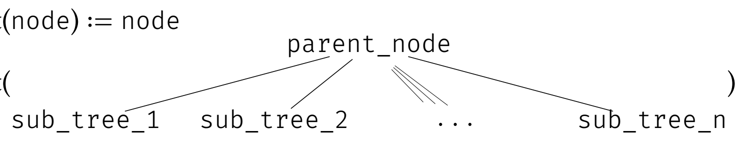
To use this module, we need to `#import "../Modules/Tree.typ": tree`

Conversion

The following function "convert" converts a tree whose nodes are Typst `content` into data that Typst can interpret as a tree.

convert : Tree TypstContent → TypstTreeData

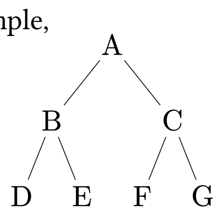
convert(`node`) ≔ `node`

convert(


)

≔ `(parent_node,convert(sub_tree_1),convert(subtree_2), … ,convert(sub_tree_n))`

For example,

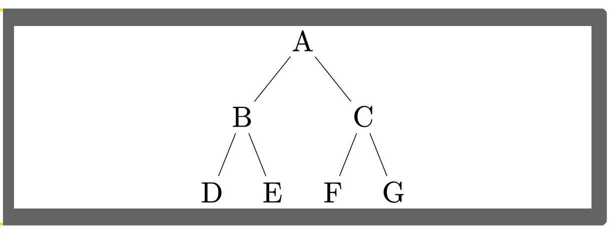


converts to `([A],([B],[D],[E]),([C],[F],[G]))`

Displaying a Tree

Once you've converted your tree, you can display it by applying the typst function `#tree()` on the data obtained upon conversion, i.e, `#tree()` : TypstTreeData → TypstTreeDisplay.

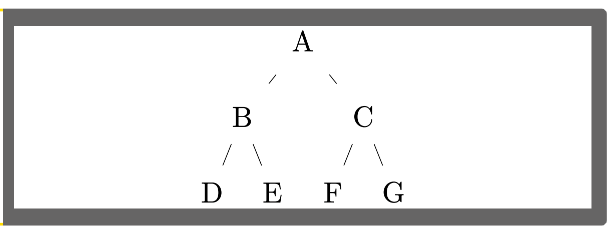
```
#tree(([A],([B],[D],[E]),([C],[F],
[G])))
```



Padding

We can control how much whitespace padding surrounds each node by setting the `pad` coefficient settable argument of the `#tree()` function,
usually to ensure that the edge does not touch the content of the node.

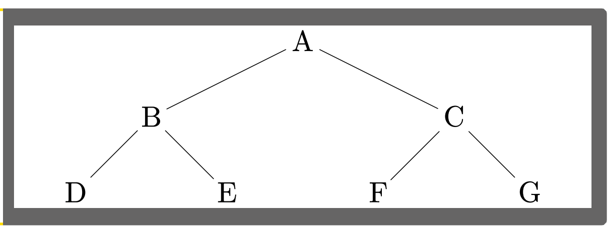
```
#tree(pad: 0.55,([A],([B],[D],[E]),
([C],[F],[G])))
```



Width of a Tree

We can control the width of a tree by the `spread` coefficient settable argument of the `#tree()` function.

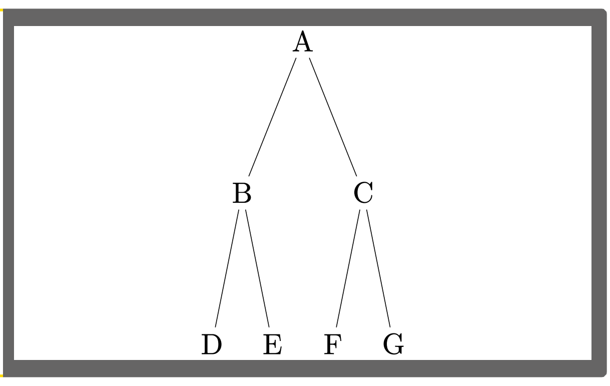
```
#tree(spread: 2,([A],([B],[D],[E]),
([C],[F],[G])))
```



Depth of a Tree

We can control the depth of a tree by the `grow` coefficient settable argument of the `#tree()` function.

```
#tree(grow: 2,([A],([B],[D],[E]),([C],
[F],[G])))
```

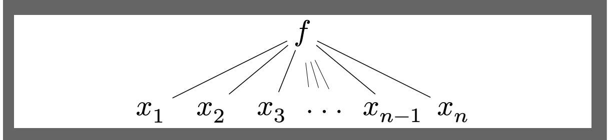


Wiiiide Trees

To do this we need to `#import "../Modules/Tree.typ": dots`.

We can suggest that a tree is very wide by making one the nodes the `#dots` function from this module.

```
#tree(
  ($f $,
    $x_1$,
    $x_2$,
    $x_3$,
    dots,
    $x_(n-1)$,
    $x_n$
  )
)
```



Deeeeep Trees

To do this we need to `#import "../Modules/Tree.typ": far_away`.

We can suggest that a tree is very wide by applying `#far_away()` function from this module on one of the nodes.

```
#tree(
  (`(:)`,
    `x1`,
    (`(:)`,
      `x2`,
      (`(:)`,
        `x3`,
        (far_away(`(:)`),
          `xn-1`,
          (`(:)`,
            `xn`,
            `[]`
          )
        )
      )
    )
  )
)
```