Recall: \texttt{type} == collection of values

Haskell’s allowable types can be defined inductively:

**Base (or primitive) types**

- \texttt{Int, Bool, Char, Float, Double, \ldots}

  Allowable values of these types are defined explicitly:
  - integers representable in 32 bits, \{\texttt{True, False}\}, 8-bit ASCII characters, \ldots

**Compound types**  Constructed from simpler types

- List types, tuple types, function types, algebraic types

  Allowable values of these types are defined inductively, in terms of values of simpler types.
Rule: If $\tau$ is a type, then $[\tau]$ is a type.

Values: The values of $[\tau]$ are:

Sequences/lists whose values all have type $\tau$

Examples:

- Values of $[\text{Int}]$ are lists of $\text{Int}$ values.

$$[5, 10, -400] \quad [777]$$

- Values of $[[\text{Int}]]$ are lists of $[\text{Int}]$ values.

$$[[5, 10, -400], \ [777]]$$

- Values of $[[[\text{Bool}]]]$ are lists of $[[\text{Bool}]]$ values.

$$[[[\text{True, False}], [\text{True}]], [[\text{False}], [\text{False, True}]])$$
Compound Types – Tuple Types

Rule: If $\tau_1, \tau_2, \ldots, \tau_n$ are all types, then $(\tau_1, \tau_2, \ldots, \tau_n)$ is a type.

Values: The values of type $(\tau_1, \tau_2, \ldots, \tau_n)$ are:

All $n$-tuples $(v_1, v_2, \ldots, v_n)$, where (for each $i \leq n$) $v_i$ is a value of type $\tau_i$.

Examples:
- Values of type $(\text{Int}, \text{Bool}, \text{Char}, [\text{Int}])$ include:
  
  $(7, \text{False}, \text{’a’}, [777])$  
  $(20, \text{True}, \text{’Z’}, [5, 10, -400])$

- Values of type $(\text{String}, (\text{Int}, \text{Bool}, \text{Char}, [\text{Int}]), \text{Bool})$ include:
  
  ("hello", (7, \text{False}, \text{’a’}, [777]), \text{True})
Compound Types – Function Types

Rule: If $\tau_1$ and $\tau_2$ are types, then $\tau_1 \rightarrow \tau_2$ is a type.

Values: The values of type $\tau_1 \rightarrow \tau_2$ are:

Functions that accept arguments of type $\tau_1$ and return results of type $\tau_2$

Examples:
- Values of type $\text{Int} \rightarrow \text{Int}$ are functions that accept an Int value and return an Int result.
- Values of type $(\text{Int}, \text{Bool}, \text{Char}, [\text{Int}]) \rightarrow [[[\text{Int}]])$ are functions that accept an $(\text{Int}, \text{Bool}, \text{Char}, [\text{Int}])$ value and return an [[[Int]]] result.
- Values of type $(\text{Int} \rightarrow \text{Bool}) \rightarrow (\text{Int} \rightarrow [[[\text{Int}]])$ are functions that accept an $\text{Int} \rightarrow \text{Bool}$ value and return an $\text{Int} \rightarrow [[[\text{Int}]])$ result.
More on Function Types

Convention: The type $\tau_1 \rightarrow (\tau_2 \rightarrow (\tau_3 \rightarrow (\cdots \rightarrow (\tau_n \rightarrow \tau) \cdots)))$ can be abbreviated as:

$$\tau_1 \rightarrow \tau_2 \rightarrow \tau_3 \rightarrow \cdots \rightarrow \tau_n \rightarrow \tau$$

Examples:

- $\text{Int} \rightarrow \text{Bool} \rightarrow \text{Char} \rightarrow \text{Int}$ is an abbreviation for
  $$\text{Int} \rightarrow (\text{Bool} \rightarrow (\text{Char} \rightarrow \text{Int}))$$
  Values of this type are functions that accept an $\text{Int}$ value and return a $\text{Bool} \rightarrow (\text{Char} \rightarrow \text{Int})$ value.

- Compare that type with $(\text{Int} \rightarrow \text{Bool}) \rightarrow (\text{Char} \rightarrow \text{Int})$: Values are functions that accept an $\text{Int} \rightarrow \text{Bool}$ value (i.e., a function) and return a $\text{Char} \rightarrow \text{Int}$ value.
Compound Types

Each of the rules we’ve seen generates more types, that feed back into one another.

Thus we have the following types:

- \([\text{Int} \rightarrow \text{Char} \rightarrow (\text{Float}, \text{Bool})]\)
  
  Values are lists of functions: each function accepts an \(\text{Int}\) value and return a \(\text{Char} \rightarrow (\text{Float}, \text{Bool})\) value.

- \((\text{Bool} \rightarrow \text{Bool}, [\text{Int}] \rightarrow \text{Char})\)
  
  Values are pairs, where for each pair:
  
  - First component is a \(\text{Bool} \rightarrow \text{Bool}\) value
  - Second component is a \([\text{Int}] \rightarrow \text{Char}\) value
Haskell’s Type System

So far: we know about values of various types.

But: Haskell programs really contain expressions (pieces of syntax), which we evaluate to obtain values. What are the types associated with expressions?

The basic idea: Suppose \( e \) is a Haskell expression, and suppose that \( e \) evaluates to a value \( v \). If \( v \) has type \( \tau \), then we want to also say that \( e \) has type \( \tau \).

The approach:

Type system \( \rightleftharpoons \) set of rules for determining the type(s) associated with expressions

Haskell’s rules are defined inductively, in a way that will seem familiar.
Haskell’s Type System : Constants

The Rule: Every Haskell built-in constant/literal has a predetermined type.

Examples:
- The constants True and False have type Bool.
- The constants "hello", "goodbye" and "" all have type String.
- The constant not has type Bool → Bool.
- The constant div has type Int → Int → Int.

Note: We’re now talking about syntax, not values, although the distinction often gets blurred.
Haskell’s Type System: Tuples

The Rule: If the expressions $e_1, e_2, \ldots, e_n$ have type $\tau_1, \tau_2, \ldots, \tau_n$ (respectively), then the expression $(e_1, e_2, \ldots, e_n)$ has type $(\tau_1, \tau_2, \cdots, \tau_n)$.

Examples:

- The expression $(\text{div } 7 \ 8, \ \text{not } \text{False})$ has type $(\text{Int}, \text{Bool})$.

- The expression $(\text{div } 7, \ \text{not } \text{False}, \ \text{not})$ has type $(\text{Int} \rightarrow \text{Int}, \ \text{Bool}, \ \text{Bool} \rightarrow \text{Bool})$. 
Haskell’s Type System: Lists

The Rule: If the expressions $e_1, e_2, \ldots, e_n$ all have type $\tau$, then the expression \([e_1, e_2, \ldots, e_n]\) has type \([\tau]\).

Examples:

- The expression \([\text{not False, False, not (not True)}]\) has type \([\text{Bool}]\).
- The expression \([\text{div 7, div 43, div (div 7 3)}]\) has type \([\text{Int} \rightarrow \text{Int}]\).
Haskell’s Type System: Function Application

The Rule: If expression \( e_1 \) has type \( \tau_1 \rightarrow \tau_2 \) and expression \( e_2 \) has type \( \tau_1 \), then the expression \( (e_1 \; e_2) \) has type \( \tau_2 \).

Convention: The expression \(( ( \cdots ( ((e_1 \; e_2) \; e_3) \; e_4) \; \cdots ) \; e_n)\) can be abbreviated as:

\[ e_1 \; e_2 \; e_3 \; \cdots \; e_n \]

Examples:
- The expression \((\text{not False})\) has type \text{Bool}.
- The expressions \((\text{div 7})\) and \text{div 7} have type \text{Int} \rightarrow \text{Int}.
- The expressions \((\text{(div 7) 8})\) and \text{div 7} \text{ 8} have type \text{Int}.
Haskell’s Type System: Function Abstractions

The Rule: If assuming that variable \( x \) has type \( \tau_1 \) lets us deduce that expression \( e \) has type \( \tau \), then the anonymous function

\[
\lambda x \rightarrow e
\]

has type \( \tau_1 \rightarrow \tau \).

Examples:

- The expression \( \lambda y \rightarrow \text{not} \ y \) has type \( \text{Bool} \rightarrow \text{Bool} \).
- The expression \( \lambda z \rightarrow \text{div} \ 7 \ z \) has type \( \text{Int} \rightarrow \text{Int} \).

Named functions are handled similarly:

\[
\text{myFun} \ z = \text{div} \ 7 \ z
\]