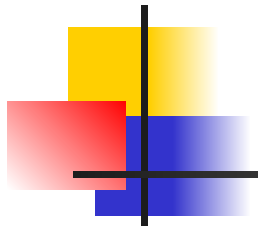


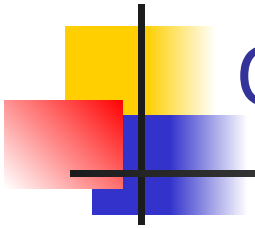
Programming Languages and Compilers (CS 421)

Talia Ringer (they/them)

4218 SC, UIUC

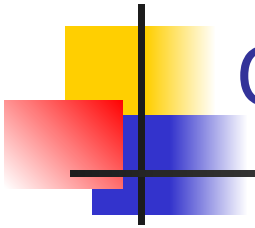
https://courses.grainger.illinois.edu/cs421/fa2023/

Based heavily on slides by Elsa Gunter, which were based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha



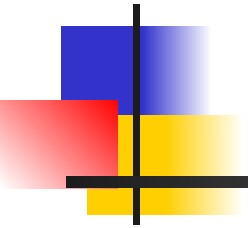
Objectives for Today

- Last class, we covered **recursive datatypes**, emphasizing how they can represent the **syntax** of **programs** for **transformations**
- We also teased **mutually recursive** and **nested recursive** datatypes
- Today, we will cover **mutually recursive** and **nested recursive** datatypes in more detail
- We will then start talking about **types** and **type checking**—another very useful thing we need to do when writing compilers and interpreters

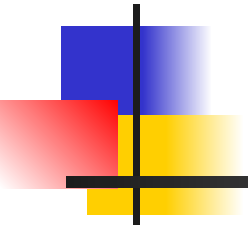


Objectives for Today

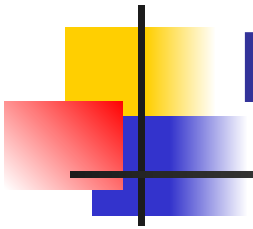
- Last class, we covered **recursive datatypes**, emphasizing how they can represent the **syntax** of **programs** for **transformations**
- We also teased **mutually recursive** and **nested recursive** datatypes
- Today, we will cover **mutually recursive** and **nested recursive** datatypes in more detail
- We will then start talking about **types** and **type checking**—another very useful thing we need to do when writing compilers and interpreters



Questions from last week?

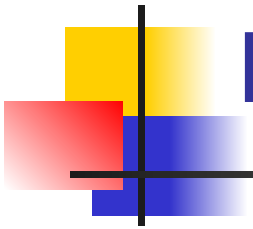


Mutually Recursive Datatypes



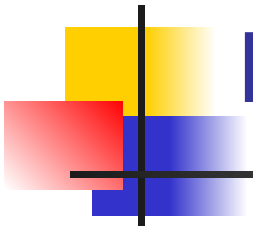
Mutually Recursive Types

```
type 'a tree =
  TreeLeaf of 'a | TreeNode of 'a treeList
and 'a treeList =
  Last of 'a tree | More of ('a tree * 'a treeList)
```



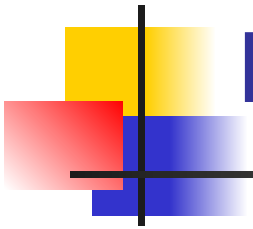
Mutually Recursive Types

```
type 'a tree =
  TreeLeaf of 'a | TreeNode of 'a treeList
and 'a treeList =
  Last of 'a tree | More of ('a tree * 'a treeList)
```



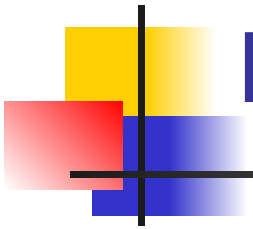
Mutually Recursive Types

```
type 'a tree =
  TreeLeaf of 'a | TreeNode of 'a treeList
and 'a treeList =
  Last of 'a tree | More of ('a tree * 'a treeList)
```



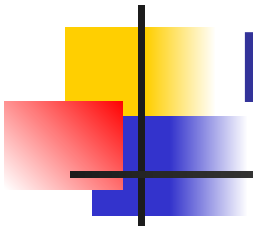
Mutually Recursive Types

```
type 'a tree =
  TreeLeaf of 'a | TreeNode of 'a treeList
and 'a treeList =
  Last of 'a tree | More of ('a tree * 'a treeList)
```



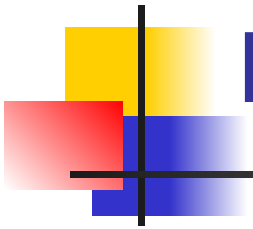
Mutually Recursive Types

```
type 'a tree =
  TreeLeaf of 'a | TreeNode of 'a treeList
and 'a treeList =
  Last of 'a tree | More of ('a tree * 'a treeList)
```



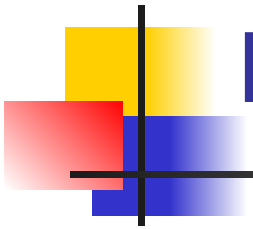
Mutually Recursive Types

```
type 'a tree =
  TreeLeaf of 'a | TreeNode of 'a treeList
and 'a treeList =
  Last of 'a tree | More of ('a tree * 'a treeList)
```



Mutually Recursive Types

```
type 'a tree =
  TreeLeaf of 'a | TreeNode of 'a treeList
and 'a treeList =
  Last of 'a tree | More of ('a tree * 'a treeList)
```

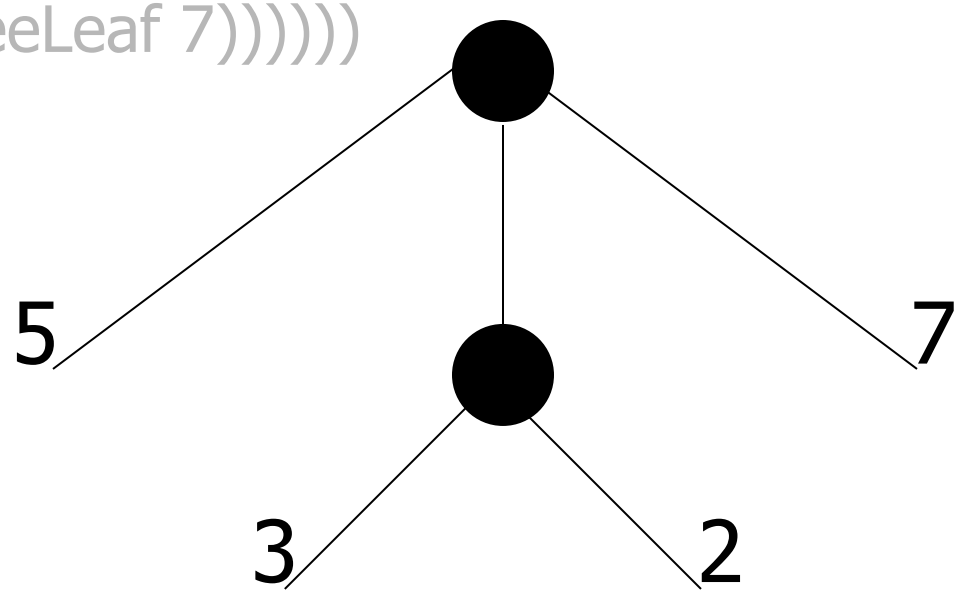


Mutually Recursive Types

```
type 'a tree =
  TreeLeaf of 'a | TreeNode of 'a treeList
and 'a treeList =
  Last of 'a tree | More of ('a tree * 'a treeList)
```

Mutually Recursive Types - Values

```
TreeNode
  (More (TreeLeaf 5,
    (More
      (TreeNode (More (TreeLeaf 3, Last (TreeLeaf 2))),
      Last (TreeLeaf 7))))))
```

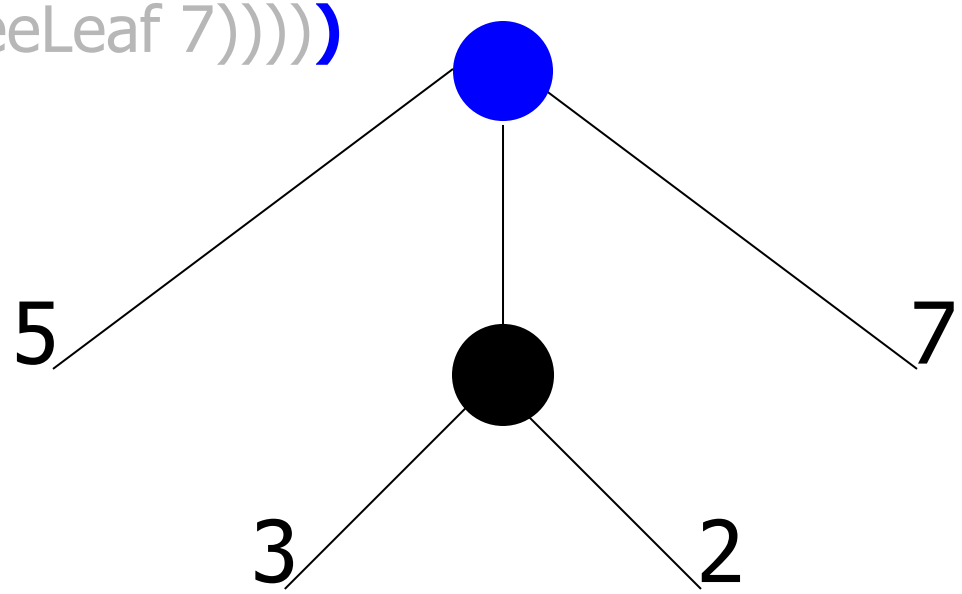


Mutually Recursive Datatypes

Mutually Recursive Types - Values

TreeNode

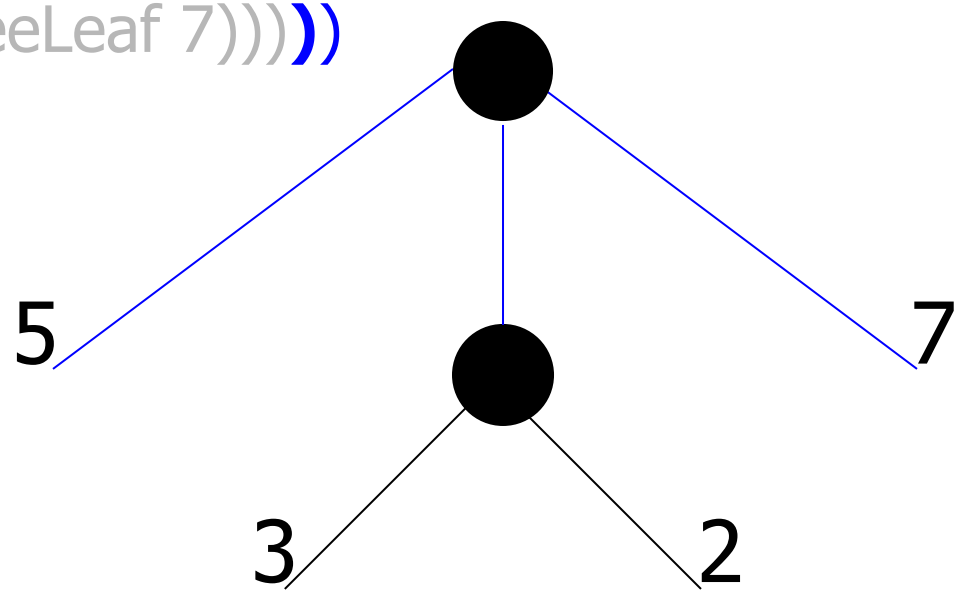
(More (TreeLeaf 5,
(More
(TreeNode (More (TreeLeaf 3, Last (TreeLeaf 2))),
Last (TreeLeaf 7))))**)**



Mutually Recursive Datatypes

Mutually Recursive Types - Values

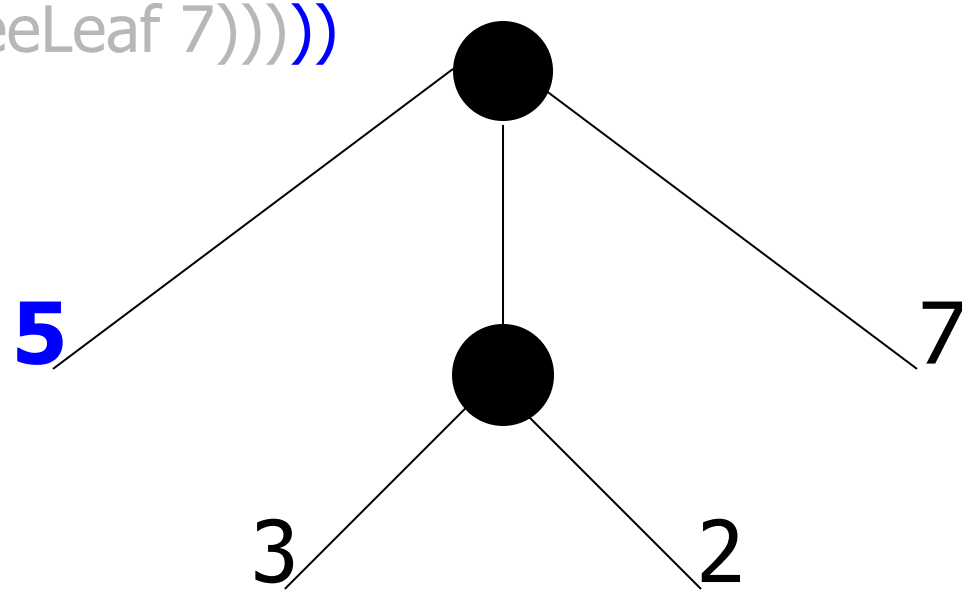
```
TreeNode
  (More (TreeLeaf 5,
    (More
      (TreeNode (More (TreeLeaf 3, Last (TreeLeaf 2))),
      Last (TreeLeaf 7)))))
```



Mutually Recursive Datatypes

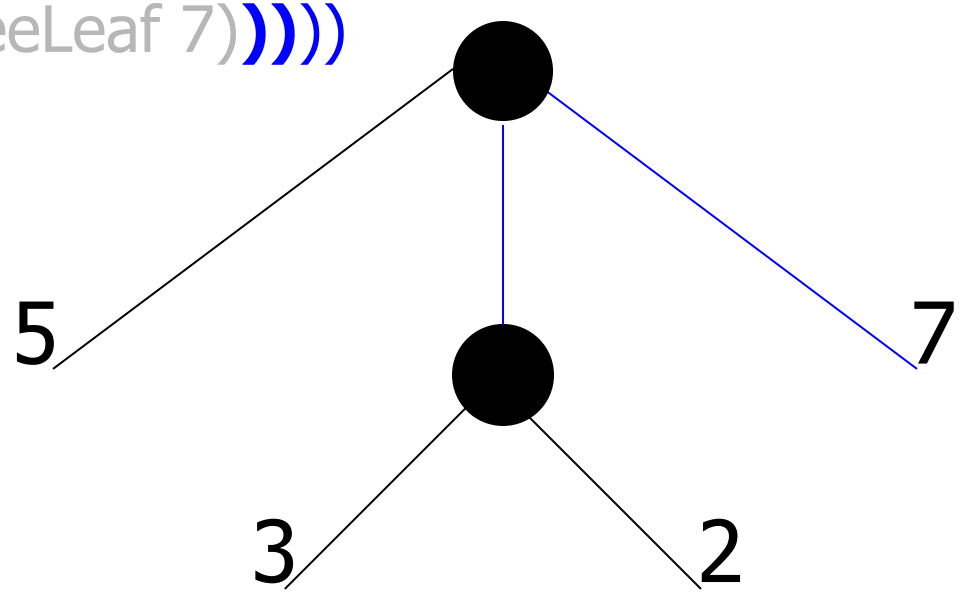
Mutually Recursive Types - Values

```
TreeNode
  (More (TreeLeaf 5,
    (More
      (TreeNode (More (TreeLeaf 3, Last (TreeLeaf 2))),
      Last (TreeLeaf 7))))
```



Mutually Recursive Types - Values

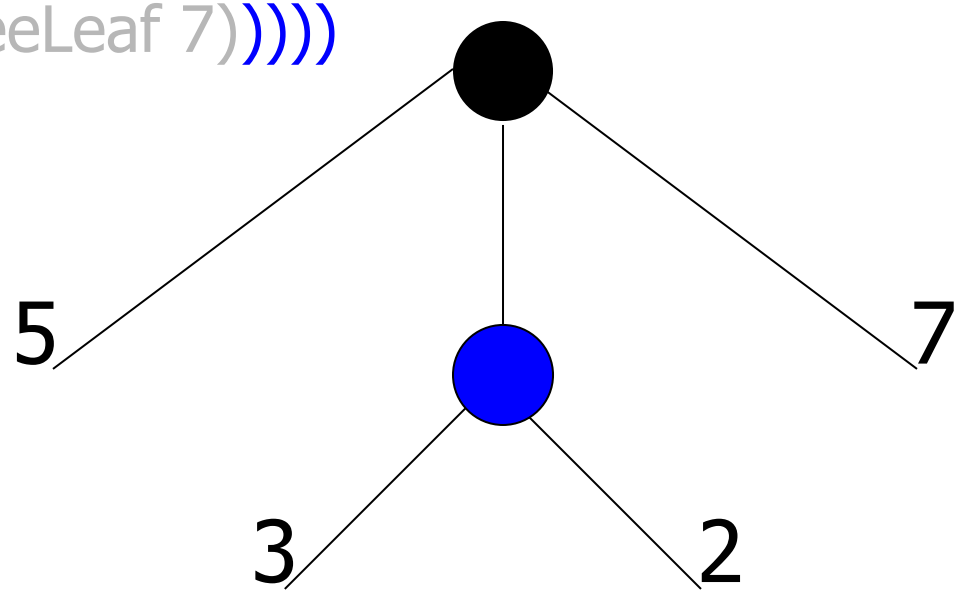
TreeNode
(More (TreeLeaf 5,
(More
(TreeNode (More (TreeLeaf 3, Last (TreeLeaf 2))),
Last (TreeLeaf 7)**)))**



Mutually Recursive Datatypes

Mutually Recursive Types - Values

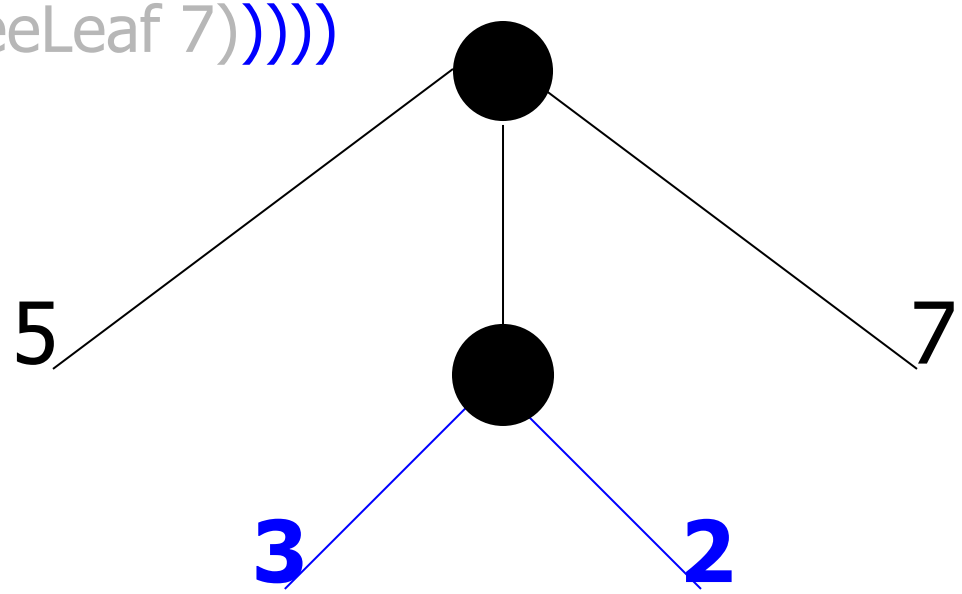
```
TreeNode
  (More (TreeLeaf 5,
    (More
      (TreeNode (More (TreeLeaf 3, Last (TreeLeaf 2)),
      Last (TreeLeaf 7))))
```



Mutually Recursive Datatypes

Mutually Recursive Types - Values

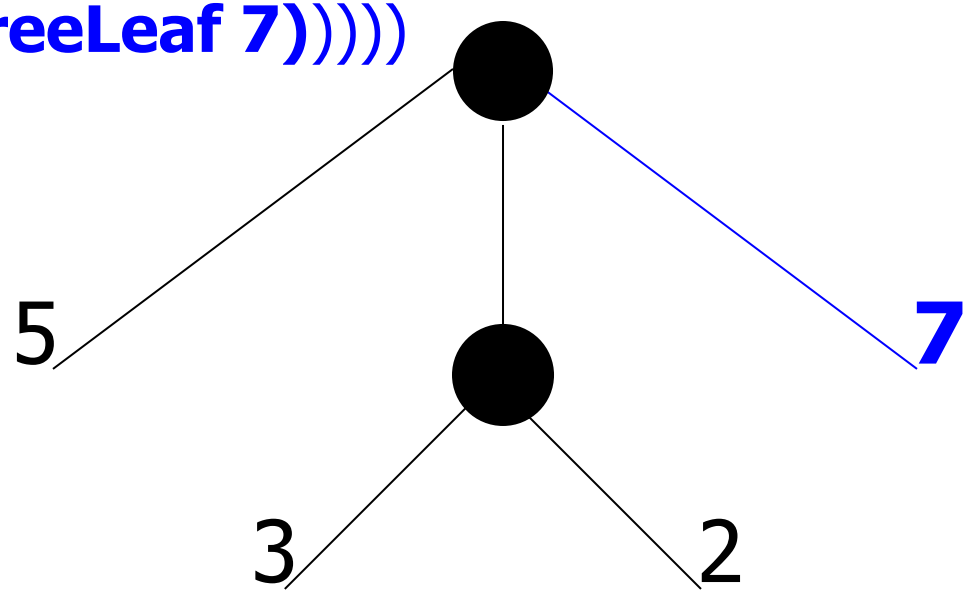
```
TreeNode
  (More (TreeLeaf 5,
    (More
      (TreeNode (More (TreeLeaf 3, Last (TreeLeaf 2))),
      Last (TreeLeaf 7)))))
```



Mutually Recursive Datatypes

Mutually Recursive Types - Values

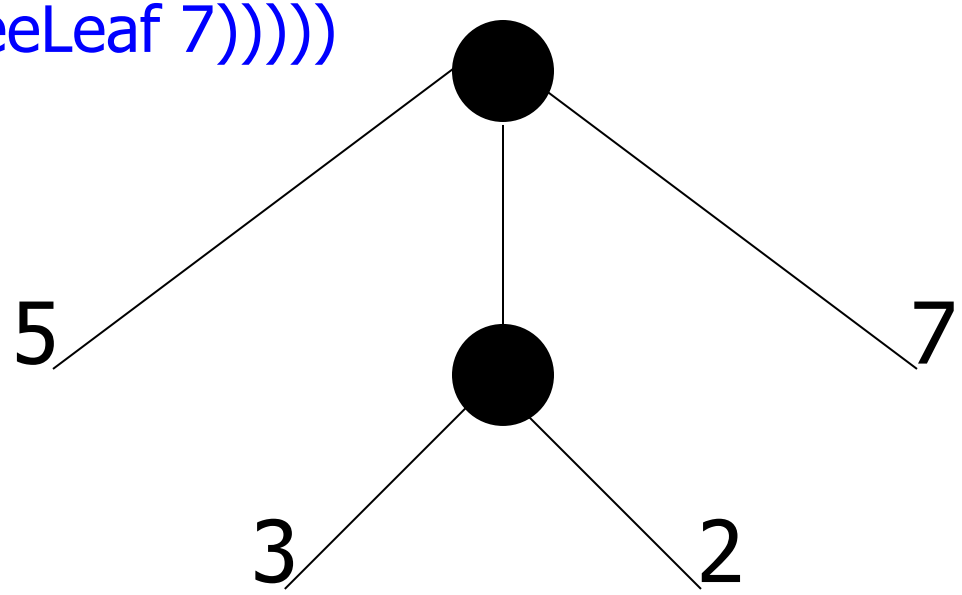
```
TreeNode
  (More (TreeLeaf 5,
    (More
      (TreeNode (More (TreeLeaf 3, Last (TreeLeaf 2))),
      Last (TreeLeaf 7)))))
```



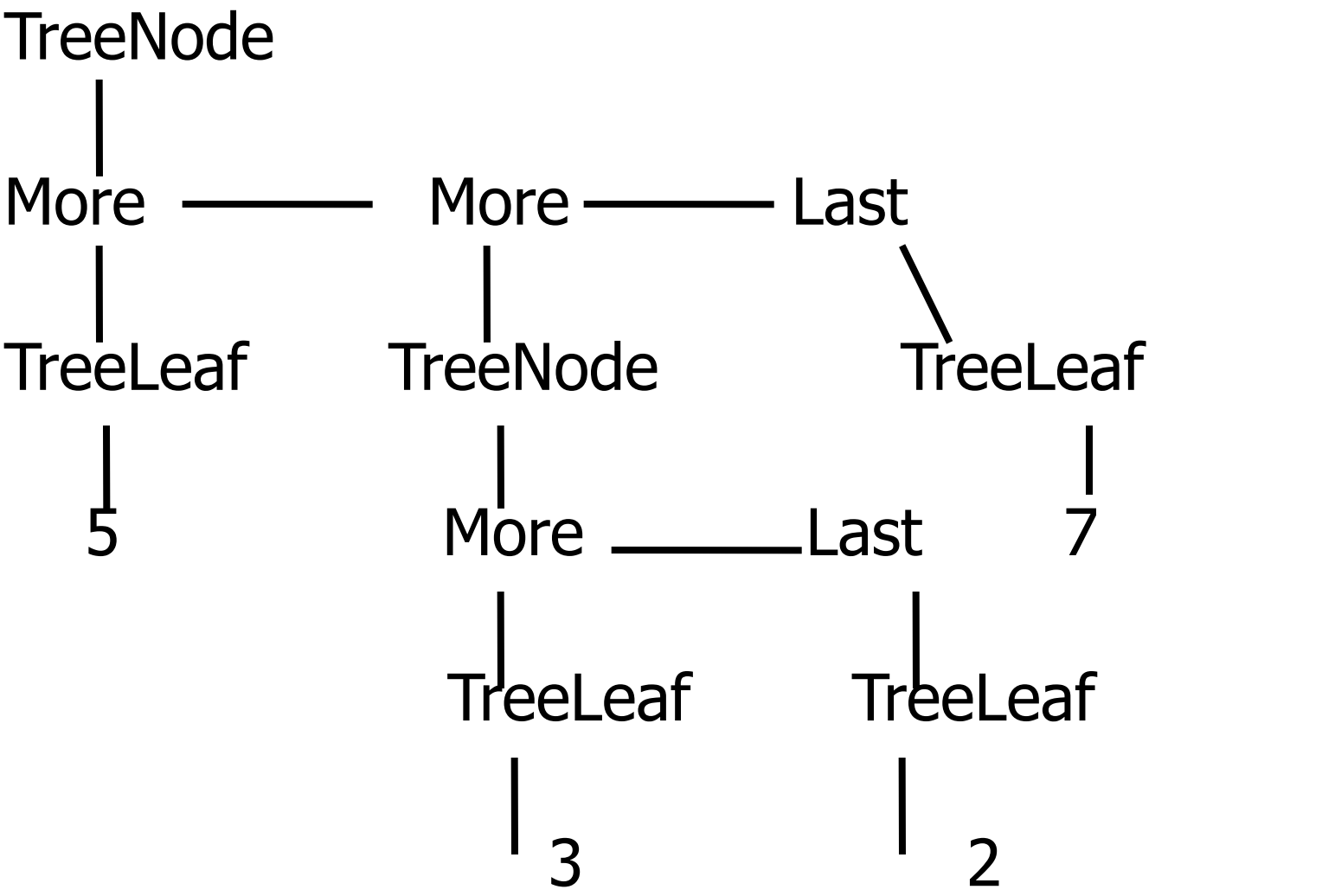
Mutually Recursive Datatypes

Mutually Recursive Types - Values

```
TreeNode
  (More (TreeLeaf 5,
    (More
      (TreeNode (More (TreeLeaf 3, Last (TreeLeaf 2))),
      Last (TreeLeaf 7)))))
```



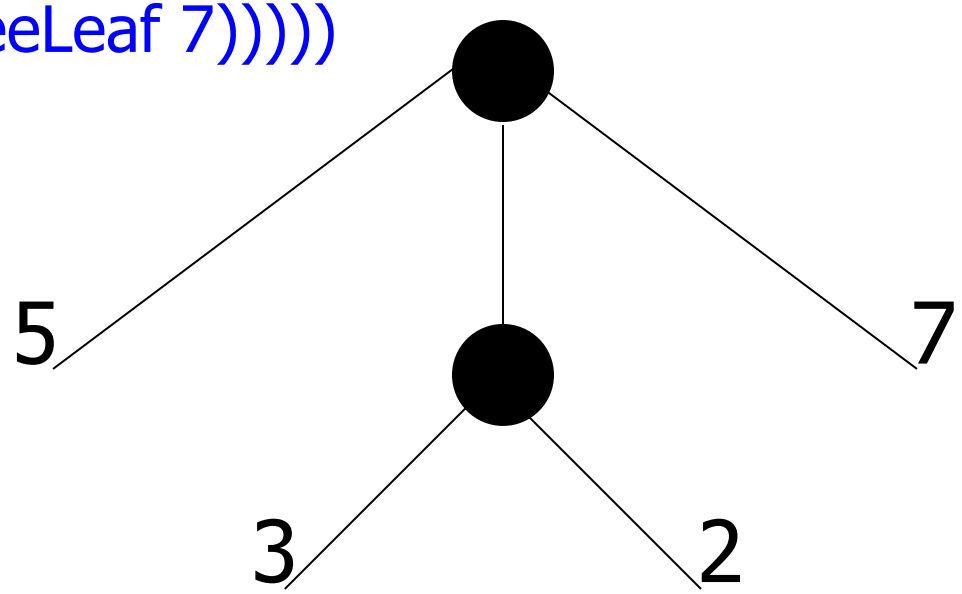
Mutually Recursive Types - Values



Mutually Recursive Datatypes

Mutually Recursive **Functions**

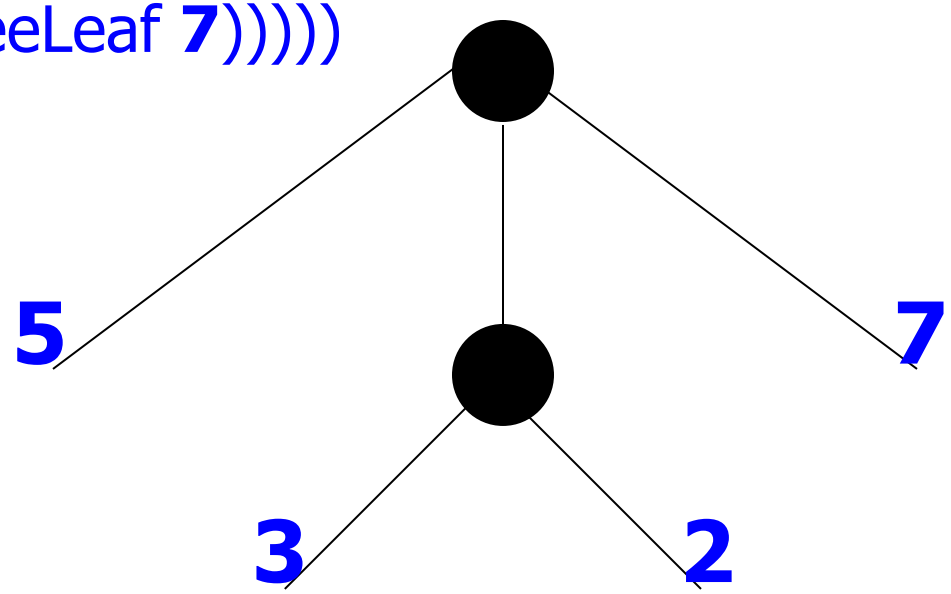
```
TreeNode
  (More (TreeLeaf 5,
    (More
      (TreeNode (More (TreeLeaf 3, Last (TreeLeaf 2))),
      Last (TreeLeaf 7)))))
```



Mutually Recursive Datatypes

Mutually Recursive **Functions**

```
TreeNode
  (More (TreeLeaf 5,
    (More
      (TreeNode (More (TreeLeaf 3, Last (TreeLeaf 2))),
      Last (TreeLeaf 7)))))
```



Mutually Recursive Datatypes

Mutually Recursive Functions

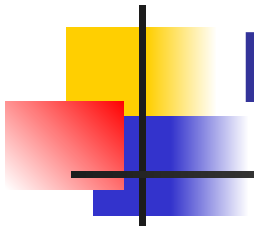
```
let rec fringe tree =
  match tree with
  | TreeLeaf x -> [x]
  | TreeNode list -> list_fringe list
and list_fringe tree_list =
  match tree_list with
  | Last tree -> fringe tree
  | More (tree, list) ->
      (fringe tree) @ (list_fringe list)
```

Mutually Recursive Datatypes

Mutually Recursive Functions

```
let rec fringe tree =
  match tree with
  | TreeLeaf x -> [x]
  | TreeNode list -> list_fringe list
and list_fringe tree_list =
  match tree_list with
  | Last tree -> fringe tree
  | More (tree, list) ->
      (fringe tree) @ (list_fringe list)
```

Mutually Recursive Datatypes



Mutually Recursive Functions

```
let rec fringe tree =
  match tree with
  | TreeLeaf x -> [x]
  | TreeNode list -> list_fringe list
and list_fringe tree_list =
  match tree_list with
  | Last tree -> fringe tree
  | More (tree, list) ->
      (fringe tree) @ (list_fringe list)
```

Mutually Recursive Datatypes

Mutually Recursive Functions

```
let rec fringe tree =
  match tree with
  | TreeLeaf x -> [x]
  | TreeNode list -> list_fringe list
and list_fringe tree_list =
  match tree_list with
  | Last tree -> fringe tree
  | More (tree, list) ->
      (fringe tree) @ (list_fringe list)
```

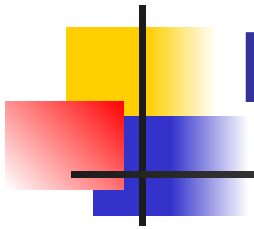
Mutually Recursive Datatypes

Mutually Recursive Functions

```
let rec fringe tree =
  match tree with
  | TreeLeaf x -> [x]
  | TreeNode list -> list_fringe list
and list_fringe tree_list =
  match tree_list with
  | Last tree -> fringe tree
  | More (tree, list) ->
      (fringe tree) @ (list_fringe list)
```

Mutually Recursive Functions

```
let rec fringe tree =
  match tree with
  | TreeLeaf x -> [x]
  | TreeNode list -> list_fringe list
and list_fringe tree_list =
  match tree_list with
  | Last tree -> fringe tree
  | More (tree, list) ->
      (fringe tree) @ (list_fringe list)
```



Mutually Recursive Functions

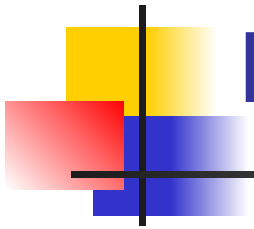
```
let rec fringe tree =
  match tree with
  | TreeLeaf x -> [x]
  | TreeNode list -> list_fringe list
and list_fringe tree_list =
  match tree_list with
  | Last tree -> fringe tree
  | More (tree, list) ->
      (fringe tree) @ (list_fringe list)
```

Mutually Recursive Functions

```
let rec fringe tree =
  match tree with
  | TreeLeaf x -> [x]
  | TreeNode list -> list_fringe list
and list_fringe tree_list =
  match tree_list with
  | Last tree -> fringe tree
  | More (tree, list) ->
      (fringe tree) @ (list_fringe list)
```

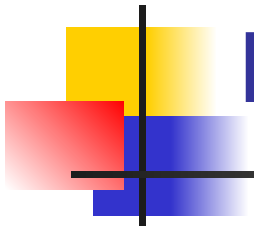
Mutually Recursive Functions

```
let rec fringe tree =
  match tree with
  | TreeLeaf x -> [x]
  | TreeNode list -> list_fringe list
and list_fringe tree_list =
  match tree_list with
  | Last tree -> fringe tree
  | More (tree, list) ->
      (fringe tree) @ (list_fringe list)
```



Mutually Recursive Functions

```
let rec fringe tree =
  match tree with
  | TreeLeaf x -> [x]
  | TreeNode list -> list_fringe list
and list_fringe tree_list =
  match tree_list with
  | Last tree -> fringe tree
  | More (tree, list) ->
      (fringe tree) @ (list_fringe list)
```

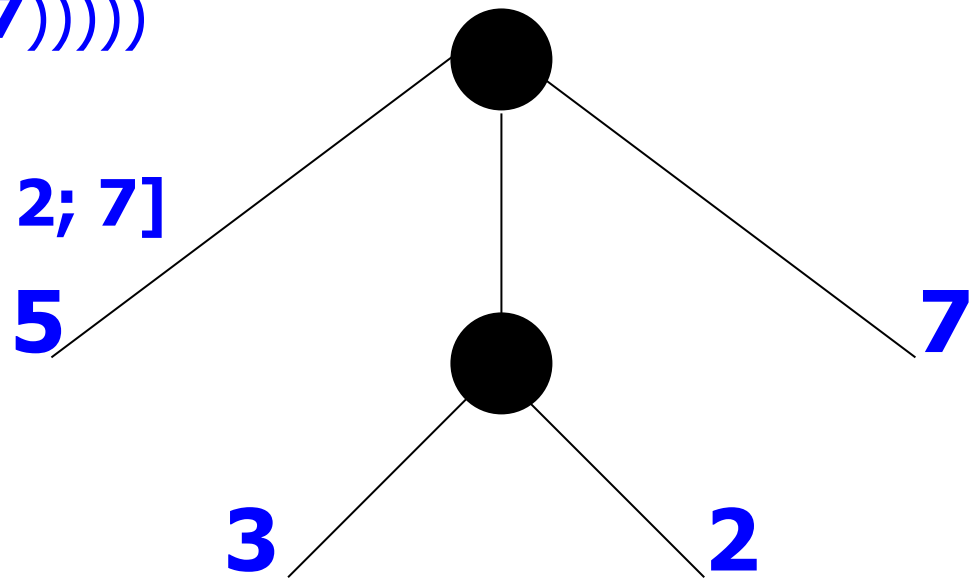


Mutually Recursive Functions

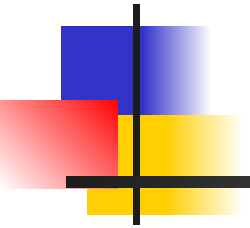
```
let rec fringe tree =
  match tree with
  | TreeLeaf x -> [x]
  | TreeNode list -> list_fringe list
and list_fringe tree_list =
  match tree_list with
  | Last tree -> fringe tree
  | More (tree, list) ->
      (fringe tree) @ (list_fringe list)
```

Mutually Recursive **Functions**

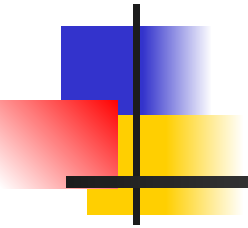
```
# let tree = TreeNode
   (More (TreeLeaf 5,
     (More
       (TreeNode (More (TreeLeaf 3, Last (TreeLeaf 2))),
       Last (TreeLeaf 7)))))
   in fringe tree;;
-   : int list = [5; 3; 2; 7]
```



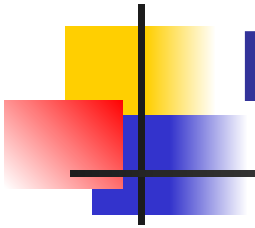
Mutually Recursive Datatypes



Questions so far?



Nested Recursive Datatypes



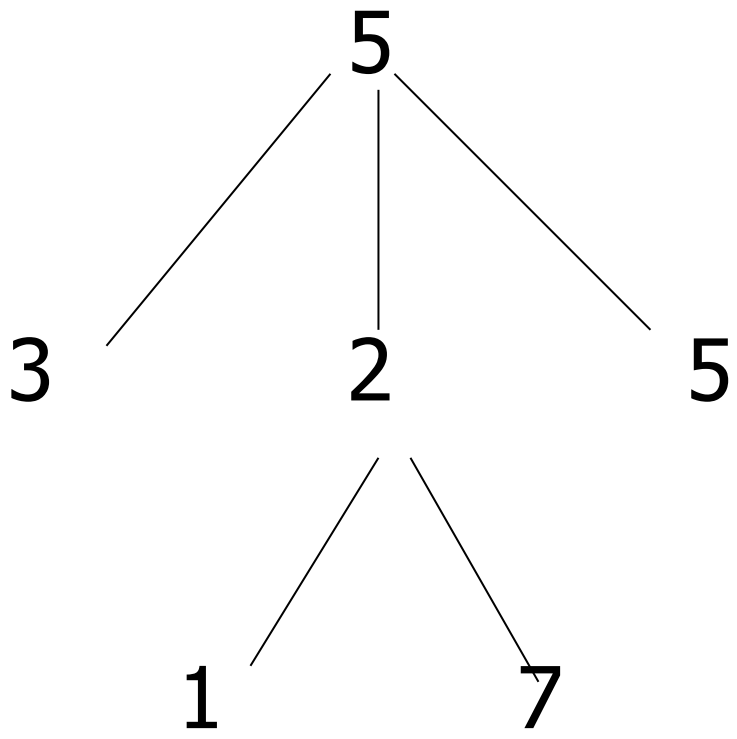
Nested Recursive Types

```
(* Alt. def, allowing empty lists & values anywhere *)
type 'a labeled_tree =
 TreeNode of ('a * 'a labeled_tree list);;
```

Nested Recursive Types - Values

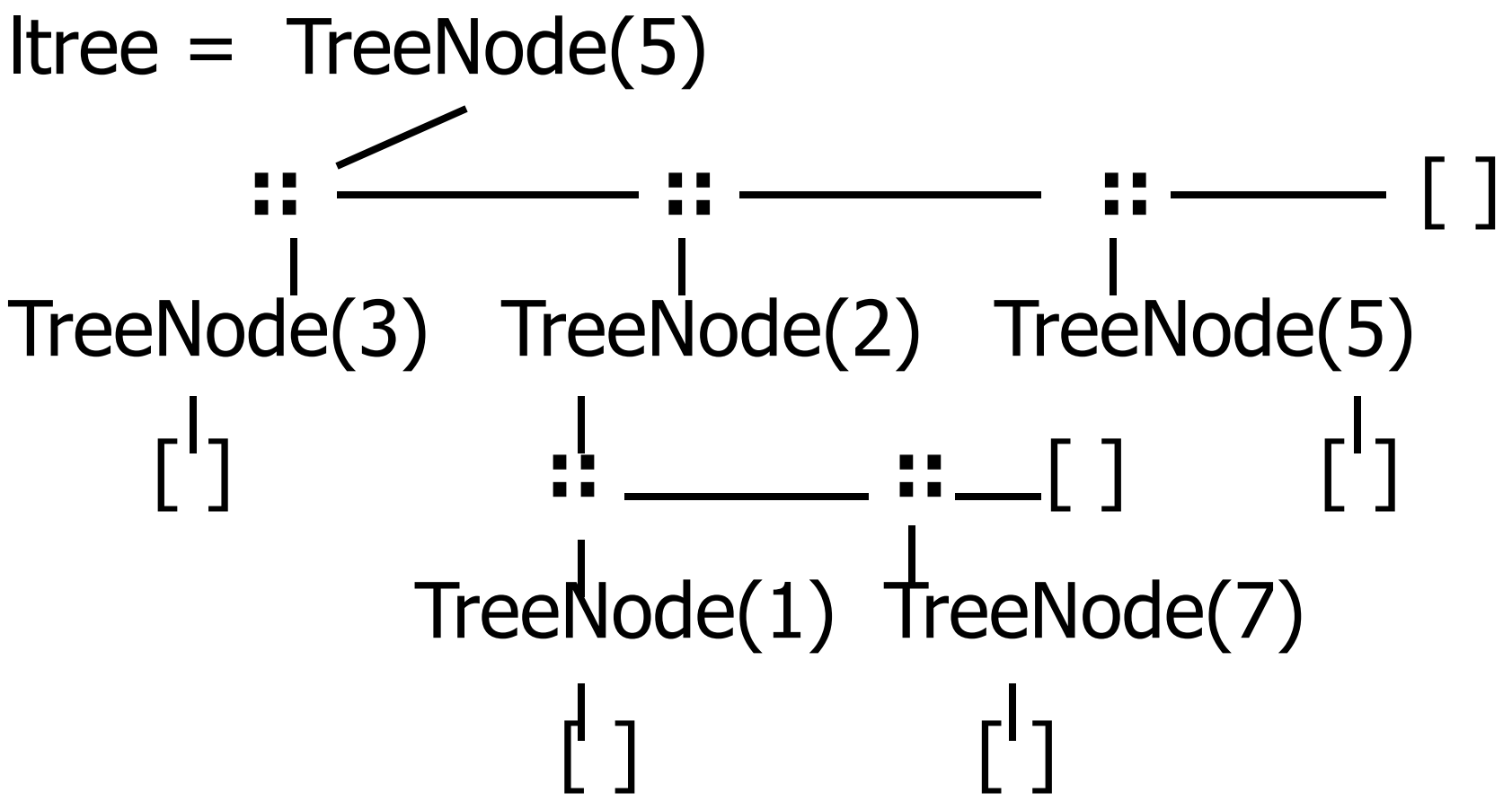
```
(* Alt. def, allowing empty lists & values anywhere *)
type 'a labeled_tree =
 TreeNode of ('a * 'a labeled_tree list);;

TreeNode
  (5,
    [TreeNode (3, []);
     TreeNode
       (2, [TreeNode (1, []); TreeNode (7, [])]);
     TreeNode (5, [])])
```

Nested Recursive Types - Values




```
TreeNode
  (5,
    [TreeNode (3, []);
     TreeNode
       (2, [TreeNode (1, []); TreeNode (7, [])]);
     TreeNode (5, [])])
```

Nested Recursive Types - Values



Mutually Recursive Functions

```
let rec flatten_tree labtree =
  match labtree with
  | TreeNode (x, ts) -> x :: flatten_tree_list ts
and flatten_tree_list ts =
  match ts with
  | [] -> []
  | labtree :: labtrees ->
    flatten_tree labtree @ flatten_tree_list labtrees
```

Mutually Recursive Functions

```
let rec flatten_tree labtree =
  match labtree with
  | TreeNode (x, ts) -> x :: flatten_tree_list ts
and flatten_tree_list ts =
  match ts with
  | [] -> []
  | labtree :: labtrees ->
    flatten_tree labtree @ flatten_tree_list labtrees
```

Nested recursive types lead to **mutually** recursive functions!

Mutually Recursive Functions

```
let rec flatten_tree labtree =
  match labtree with
  | TreeNode (x, ts) -> x :: flatten_tree_list ts
and flatten_tree_list ts =
  match ts with
  | [] -> []
  | labtree :: labtrees ->
    flatten_tree labtree @ flatten_tree_list labtrees
```

Nested recursive types lead to **mutually** recursive functions!

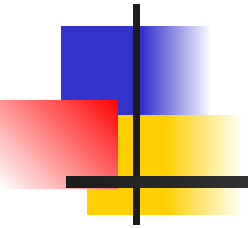
Can get around if clever, but **nontrivial**.

Mutually Recursive Functions

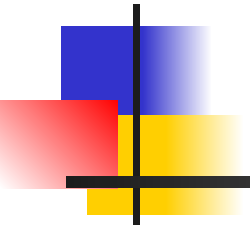
```
let rec flatten_tree labtree =
  match labtree with
  | TreeNode (x, ts) -> x :: flatten_tree_list ts
and flatten_tree_list ts =
  match ts with
  | [] -> []
  | labtree :: labtrees ->
    flatten_tree labtree @ flatten_tree_list labtrees
```

Nested recursive types lead to **mutually** recursive functions!

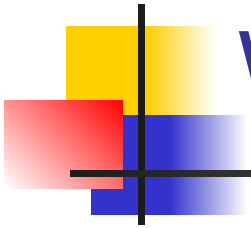
And we need **polymorphism** to work around!



Questions so far?



Types and Type Checking



Why Types?

- Types play a key role in:
  - **Data abstraction** in the design of programs
    - Keeping track of important information for you
    - Abstracting away irrelevant details
  - **Type checking** in the analysis of programs
    - e.g., ruling out entire classes of bugs
  - **Compile-time code generation** in the translation and execution of programs
    - Data layout (how many words; which are data and which are pointers) dictated by type

No Really, Why Types?

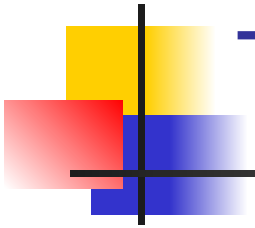
- https://www.destroyallsoftware.com/talks/wat

Terminology

- Type: A **type** T defines possible data values
  - For the sake of this class, it's enough to imagine it as being a **set** of possible data values
  - e.g., short in C is $\{x \mid 2^{15} - 1 \geq x \geq -2^{15}\}$
  - A value (or term) in this set is said to have type T
- **Type system**: rules of a language assigning types to expressions
  - One can view a type system as *ruling out possibly "bad" expressions* in a language
  - Deeply and beautifully connected to logics

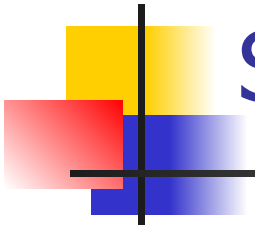
Terminology

- Type: A **type** T defines possible data values
  - For the sake of this class, it's enough to imagine it as being a **set** of possible data values
  - e.g., short in C is $\{x \mid 2^{15} - 1 \geq x \geq -2^{15}\}$
  - A value (or term) in this set is said to have type T
- **Type system**: rules of a language assigning types to expressions
  - One can view a type system as *ruling out possibly "bad" expressions* in a language
  - Deeply and beautifully connected to logics



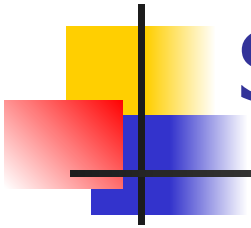
Types as Specifications

- Types describe **properties** of programs
- Different type systems describe **different properties**, e.g.,
  - Data is read-write versus read-only
  - Operation has authority to access data
  - Data came from "right" source
- With fancy types, can prove **theorems** by writing programs whose types represent those theorems
- **Common type systems** focus on types describing same data layout and access properties



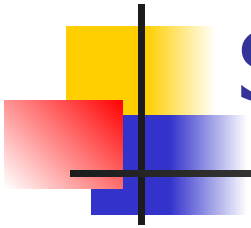
Sound Type System

- A type system is **sound** if in that system, whenever an expression is assigned type T, and it evaluates to value v, then v is in the set of values defined by T
- Informally, if the type checker says a term has a given type, then when you actually run the program it's going to have that type still, no matter what weird thing you do to the term
- OCaml, Scheme, and Rust have sound type systems
- Most implementations of C and C++ do not



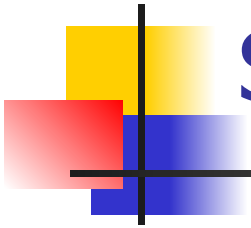
Strongly Typed Language

- When no application of an operator to arguments can lead to a runtime type error, the language is said to be **strongly typed**
  - Eg: 1 + 2.3;;
- What this actually implies depends on the definition of "type error," which varies by language



Strongly Typed Language

- C++ claimed to be "strongly typed", but
  - Union types allow creating a value at one type and using it at another
  - Type coercions may cause unexpected (undesirable) effects
  - No array bounds check (in fact, no runtime checks at all)
- SML, OCaml "strongly typed" but still must do dynamic array bounds checks, runtime type case analysis, and other checks
- Coq, Lean, Agda, Idris can do really fancy checks



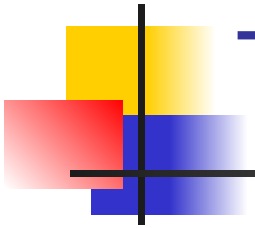
Static vs. Dynamic Types

- **Static type**: type assigned to an expression at compile time
- **Dynamic type**: type assigned to a storage location at run time
- **Statically typed language**: static type assigned to every expression at compile time
- **Dynamically typed language**: type of an expression determined at run time
- **Gradually typed language:** continuum of languages between dynamic and static typing

Static vs. Dynamic Types

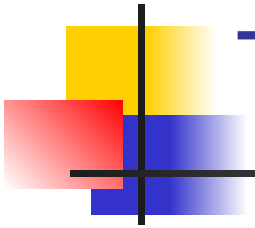
- **Static type**: type assigned to an expression at compile time
- **Dynamic type**: type assigned to a storage location at run time
- **Statically typed language**: static type assigned to every expression at compile time
- **Dynamically typed language**: type of an expression determined at run time
- **Gradually typed language:** continuum of languages between dynamic and static typing

Gradual types are not explicitly covered in class



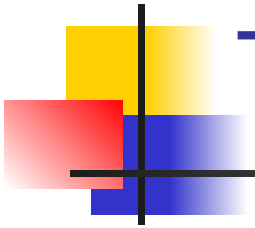
Type Checking

- When is op(arg1 , … , argn) allowed?
- **Type checking** assures operations are applied to the right number of arguments of the right types
  - "Right type" may mean same type as was **specified**, or may mean that there is a predefined implicit **coercion** that will be applied
- Used to resolve overloaded operations



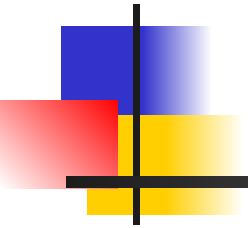
Type Checking

- When is op(arg1 , … , argn) allowed?
- **Type checking** assures operations are applied to the right number of arguments of the right types
  - "Right type" may mean same type as was **specified**, or may mean that there is a predefined implicit **coercion** that will be applied
- Used to resolve overloaded operations

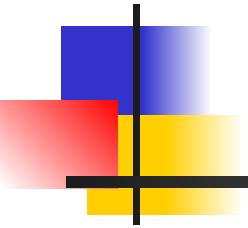


Type Declarations & Type Inference

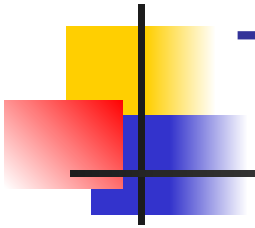
- **Type declarations**: explicit assignment of types to terms in source code
  - Must be checked in a strongly typed language
  - Often not necessary for strong typing or even static typing (depends on the type system)
- **Type inference**: a program analysis to assign a type to a term in its context
  - Fully static type inference first introduced by Robin Miller in ML
  - Haskell, OCaml, SML all use type inference
    - Records are a problem for type inference



Questions so far?

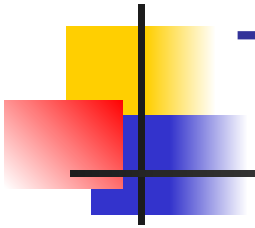


Types and **Type Checking**



Type Checking

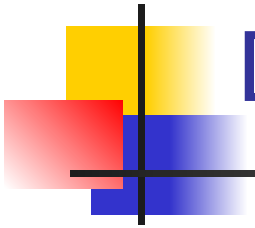
- When is op(arg1 , … , argn) allowed?
- **Type checking** assures operations are applied to the right number of arguments of the right types
  - "Right type" may mean same type as was **specified**, or may mean that there is a predefined implicit **coercion** that will be applied
- Used to resolve overloaded operations



Type Checking

- Type checking may be done **statically** at compile time or **dynamically** at run time
- Dynamically typed languages (e.g., LISP, Prolog) do only dynamic type checking
- Statically typed languages can do most type checking statically
- Real life does not like binary discrete categories of things so much (consider Python with mypy)

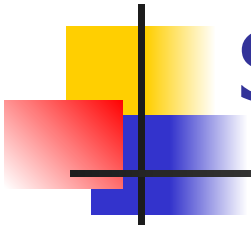
Dynamic Type Checking



- **Dynamic type checking** is performed at *run-time* before each operation is applied
- Types of variables and operations left unspecified until run-time
  - Same variable may be used at different types
- Data object must contain type information
- Errors aren't detected until violating application is executed (maybe *years* after the code was written)

Dynamic Type Checking

- **Dynamic type checking** is performed at *run-time* before each operation is applied
- Types of variables and operations left unspecified until run-time
  - Same variable may be used at different types
- Data object must contain type information
- Errors aren't detected until violating application is executed (maybe *years* after the code was written)



Static Type Checking

- **Static type checking** is performed after parsing, before code generation
- Type of every variable and signature of every operator must be known at compile time
- Can eliminate need to store type information in data object if no dynamic type checking is needed
- Catches many programming errors at earliest point
- Can't check types that depend on dynamically computed values
  - e.g., array bounds, *unless* your type system is very fancy (dependent types)

Static Type Checking

- **Static type checking** is performed after parsing, before code generation
- Type of every variable and signature of every operator must be known at compile time
- Can eliminate need to store type information in data object if no dynamic type checking is needed
- Catches many programming errors at earliest point
- Can't check types that depend on dynamically computed values
  - e.g., array bounds, *unless* your type system is very fancy (dependent types)

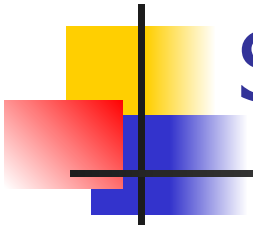
Static Type Checking

- **Static type checking** is performed after parsing, before code generation
- Type of every variable and signature of every operator must be known at compile time
- Can eliminate need to store type information in data object if no dynamic type checking is needed
- Catches many programming errors at earliest point
- Can't check types that depend on dynamically computed values
  - e.g., array bounds, *unless* your type system is very fancy (dependent types)

Static Type Checking

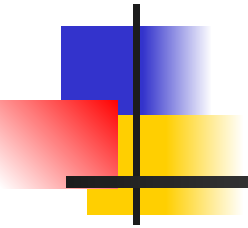
- **Static type checking** is performed after parsing, before code generation
- Type of every variable and signature of every operat
- Can e in data c eeded
- Catche t point
- Can't check types that depend on dynamically computed values
  - e.g., array bounds, *unless* your type system is very fancy (**dependent types**)

Dependent types are not explicitly covered in class, but I'm obsessed with them, so please ask in office hours or something



Static Type Checking

- Typically places **restrictions** on languages
  - Garbage collection, usually **(except Rust!)**
  - References instead of pointers **(Rust has both!)**
  - All variables initialized when created
  - Variable only used at one type
    - Union types allow for work-arounds, but effectively introduce dynamic type checks



Type Judgments

Format of Type Judgments

- A **type judgement** has the form **$\Gamma \vdash t : T$**
- Informally: "in gamma, t has type T"
- Γ (Γ in latex) is a **typing environment**
  - Maps terms (variables, and function names when function names are not variables) to types
  - Γ is a set of the form $\{ t_1 : T_1 , \ldots , t_n : T_n \}$
  - For any t_i at most one T_i such that $(t_i : T_i \in \Gamma)$
- t is a **term** (program expression)
- T is a **type** to be assigned to t
- $\vdash$ pronounced "**turnstile**" or "entails" ($\vdash$)

Format of Type Judgments

- A **type judgement** has the form **$\Gamma \vdash t : T$**
- Informally: "in gamma, t has type T"
- Γ (Γ in latex) is a **typing environment**
  - Maps terms (variables, and function names when function names are not variables) to types
  - Γ is a set of the form $\{ t_1 : T_1 , \ldots , t_n : T_n \}$
  - For any t_i at most one T_i such that $(t_i : T_i \in \Gamma)$
- t is a **term** (program expression)
- T is a **type** to be assigned to t
- $\vdash$ pronounced "**turnstile**" or "entails" ($\vdash$)

Format of Type Judgments

- A **type judgement** has the form **$\Gamma \vdash t : T$**
- Informally: "in gamma, t has type T"
- Γ (Γ in latex) is a **typing environment**
  - Maps terms (variables, and function names when function names are not variables) to types
  - Γ is a set of the form $\{ t_1 : T_1 , \ldots , t_n : T_n \}$
  - For any t_i at most one T_i such that $(t_i : T_i \in \Gamma)$
- t is a **term** (program expression)
- T is a **type** to be assigned to t
- $\vdash$ pronounced "**turnstile**" or "entails" ($\vdash$)

Format of Type Judgments

- A **type judgement** has the form **Γ ⊢ t : T**
- Informally: "in gamma, t has type T"
- Γ (Γ in latex) is a **typing environment**
  - Maps terms (variables, and function names when function names are not variables) to types
  - Γ is a set of the form $\{ t_1 : T_1 , \ldots , t_n : T_n \}$
  - For any t_i at most one T_i such that $(t_i : T_i \in \Gamma)$
- t is a **term** (program expression)
- T is a **type** to be assigned to t
- ⊢ pronounced "**turnstile**" or "entails" ($\vdash$)

Axioms – Constants (Monomorphic)

$$\frac{}{\Gamma \vdash n : \text{int}}$$ (assuming n is an integer constant)

$$\frac{}{\Gamma \vdash \text{true} : \text{bool}} \qquad \frac{}{\Gamma \vdash \text{false} : \text{bool}}$$

- These rules are true with any typing environment
- Γ, n are metavariables

Axioms – Variables (Monomorphic Rule)

Notation: Let $\Gamma(v) = T$ if $v : T \in \Gamma$

Note: if such T exits, its unique

Variable axiom:

$$\frac{}{\Gamma \vdash v : T} \quad \text{if } \Gamma(v) = T$$

Simple Rules – Arithmetic (Mono)

Primitive Binary operators ($\oplus \in \{ +, -, *, \ldots\}$):

$$\frac{\Gamma \vdash t_1 : T_1 \quad \Gamma \vdash t_2 : T_2 \quad (\oplus) : T_1 \rightarrow T_2 \rightarrow T_3}{\Gamma \vdash t_1 \oplus t_2 : T_3}$$

Special case: Relations ($\sim \in \{ <, >, =, <=, >= \}$):

$$\frac{\Gamma \vdash t_1 : T \quad \Gamma \vdash t_2 : T \quad (\sim) : T \rightarrow T \rightarrow \text{bool}}{\Gamma \vdash t_1 \sim t_2 : \text{bool}}$$

For the moment, think T is int

Type Judgments

Simple Rules – Arithmetic (Mono)

Primitive Binary operators ($\oplus \in \{ +, -, *, \ldots\}$):

$$\frac{\Gamma \vdash t_1 : T_1 \quad \Gamma \vdash t_2 : T_2 \quad (\oplus) : T_1 \rightarrow T_2 \rightarrow T_3}{\Gamma \vdash t_1 \oplus t_2 : T_3}$$

Special case: Relations ($\sim \in \{ <, >, =, <=, >= \}$):

$$\frac{\Gamma \vdash t_1 : T \quad \Gamma \vdash t_2 : T \quad (\sim) : T \rightarrow T \rightarrow \text{bool}}{\Gamma \vdash t_1 \sim t_2 : \text{bool}}$$

For the moment, think T is int

Type Judgments

Simple Rules – Arithmetic (Mono)

Primitive Binary operators ($\oplus \in \{ +, -, *, \ldots\}$):

$$\frac{\Gamma \vdash t_1 : T_1 \quad \Gamma \vdash t_2 : T_2 \quad (\oplus) : T_1 \rightarrow T_2 \rightarrow T_3}{\Gamma \vdash t_1 \oplus t_2 : T_3}$$

Special case: Relations ($\sim \in \{ <, >, =, <=, >= \}$):

$$\frac{\Gamma \vdash t_1 : T \quad \Gamma \vdash t_2 : T \quad (\sim) : T \rightarrow T \rightarrow \text{bool}}{\Gamma \vdash t_1 \sim t_2 : \text{bool}}$$

For the moment, think T is int

Type Judgments

Example: { x : int } ⊢ x + 2 = 3 : bool

What do we need to show first?

$$\frac{???}{\{x : int\} \vdash x + 2 = 3 : bool}$$

Example: { x : int } ⊢ x + 2 = 3 : bool

What do we need to show first?

$$\frac{\{x : int\} \vdash x + 2 : int \qquad \{x : int\} \vdash 3 : int}{\{x : int\} \vdash x + 2 = 3 : bool}\ Bin$$

Example: { x : int } ⊢ x + 2 = 3 : bool

Left-hand side?

$$\frac{\dfrac{???}{\{x : int\} \vdash x + 2 : int} \qquad \{x : int\} \vdash 3 : int}{\{x : int\} \vdash x + 2 = 3 : bool}\ Bin$$

Example: { x : int } ⊢ x + 2 = 3 : bool

Left-hand side?

$$\dfrac{\dfrac{\{x : int\} \vdash x : int \quad \{x : int\} \vdash 2 : int}{\{x : int\} \vdash x + 2 : int}\,\text{Bin} \qquad \{x : int\} \vdash 3 : int}{\{x : int\} \vdash x + 2 = 3 : bool}\,\text{Bin}$$

Example: { x : int } ⊢ x + 2 = 3 : bool

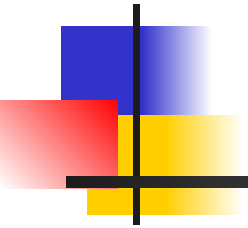
How to finish?

$$\dfrac{\dfrac{\{x : int\} \vdash x : int \quad \{x : int\} \vdash 2 : int}{\{x : int\} \vdash x + 2 : int}\,Bin \qquad \{x : int\} \vdash 3 : int}{\{x : int\} \vdash x + 2 = 3 : bool}\,Bin$$

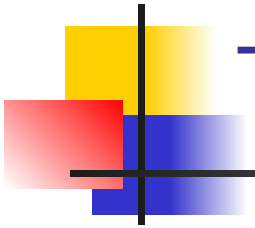
Example: { x : int } ⊢ x + 2 = 3 : bool

Complete proof (type derivation)

$$\dfrac{\dfrac{\dfrac{}{\{x : int\} \vdash x : int}\,Var \quad \dfrac{}{\{x : int\} \vdash 2 : int}\,Const}{\{x : int\} \vdash x + 2 : int}\,Bin \quad \dfrac{}{\{x : int\} \vdash 3 : int}\,Const}{\{x : int\} \vdash x + 2 = 3 : bool}\,Bin$$

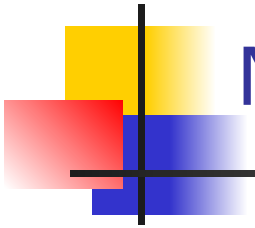


Questions?



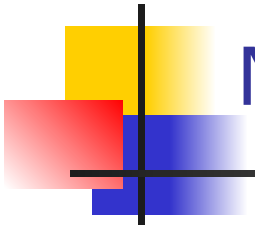
Takeaways

- We saw **mutual** and **nested** recursive datatypes in more detail than last time. Both lead to **mutually recursive** functions.
- It's possible to **work around** mutual recursion if you want—thanks to **higher-order functions** and **polymorphism**.
- **Types** can be useful for many things.
- **Γ ⊢ t : T** means that a term **t** has type **T** in type conte **Γ**.
- Such a **judgment** can be **checked statically** or **dynamically** (or, IRL, sometimes a mix).



Next Class: More Type Checking

- We saw **mutual** and **nested** recursive datatypes in more detail than last time. Both lead to **mutually recursive** functions.
- It's possible to **work around** mutual recursion if you want—thanks to **higher-order functions** and **polymorphism**.
- **Types** can be useful for many things.
- **Γ ⊢ t : T** means that a term **t** has type **T** in type conte **Γ**.
- Such a **judgment** can be **checked statically** or **dynamically** (or, IRL, sometimes a mix).



Next Class

- **EC1 graded!**
  - It's really hard to catch bugs in language-model-generated code! (~25% missed bugs in final generated code that I caught)
  - Also impacted me when I tried it … traditional expertise doesn't translate directly here
- **EC2 is late, but coming**
- **WA4** will be due Thursday
- **Quiz 3** on **MP5** is next Tuesday
- All deadlines can be found on **course website**
- Use **office hours** and **class forums** for help