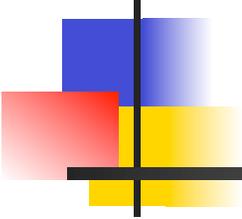


Programming Languages and Compilers (CS 421)

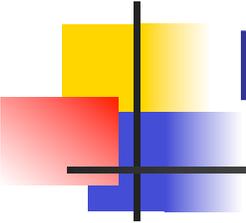


Elsa L Gunter

2112 SC, UIUC

<http://courses.engr.illinois.edu/cs421>

Based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha



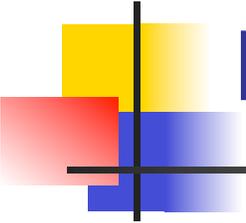
Mutually Recursive Types

```
# type 'a tree = TreeLeaf of 'a  
  | TreeNode of 'a treeList
```

```
and 'a treeList = Last of 'a tree  
  | More of ('a tree * 'a treeList);;
```

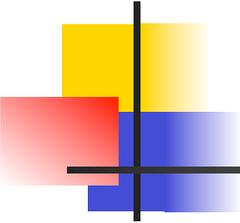
```
type 'a tree = TreeLeaf of 'a | TreeNode of 'a  
  treeList
```

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



Mutually Recursive Types - Values

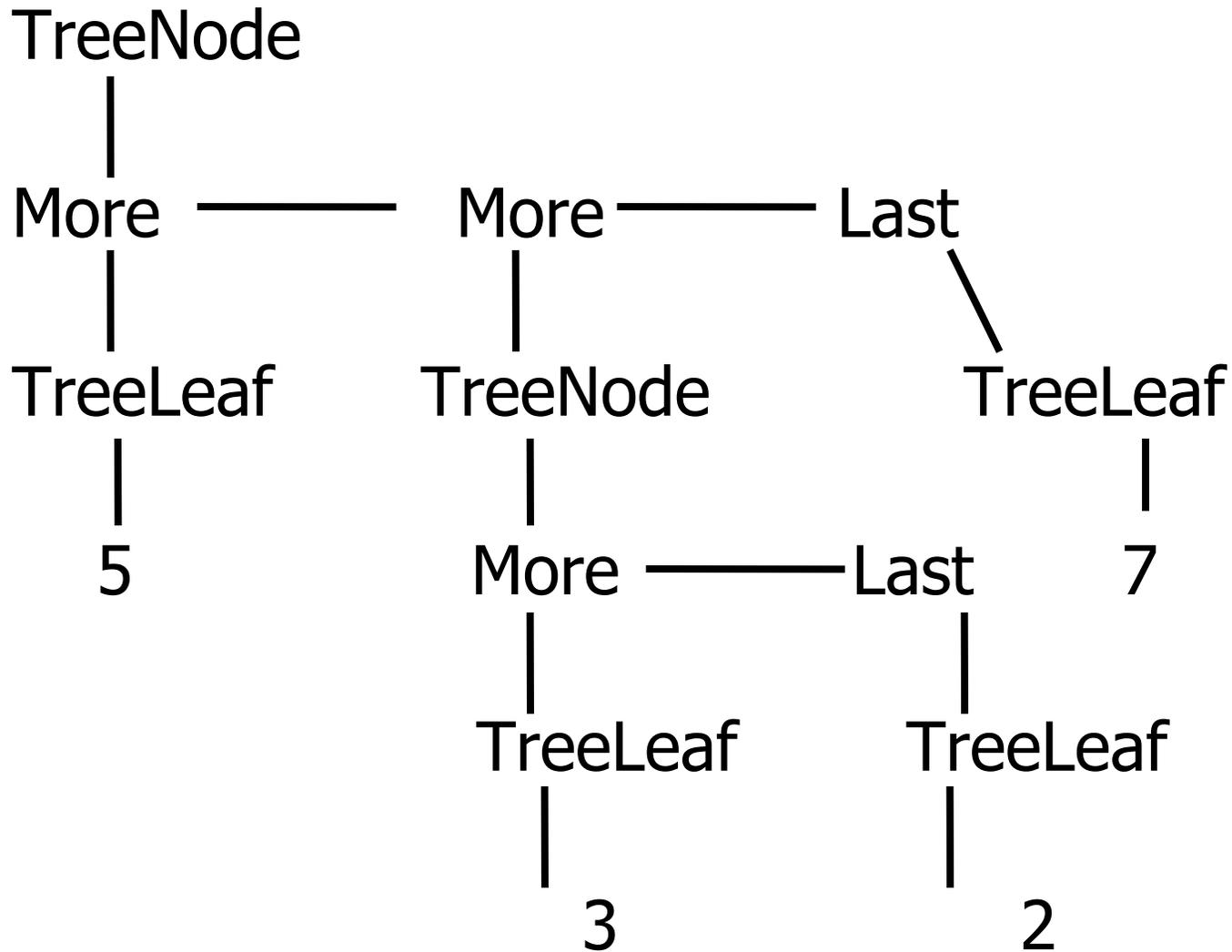
```
# let tree =  
  TreeNode  
    (More (TreeLeaf 5,  
          (More (TreeNode  
                (More (TreeLeaf 3,  
                      Last (TreeLeaf 2))),  
                      Last (TreeLeaf 7))))));;
```

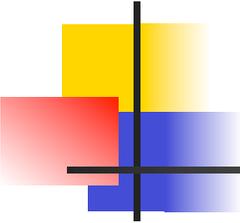


Mutually Recursive Types - Values

```
val tree : int tree =  
  TreeNode  
    (More  
      (TreeLeaf 5,  
        More  
          (TreeNode (More (TreeLeaf 3, Last  
            (TreeLeaf 2))), Last (TreeLeaf 7))))))
```

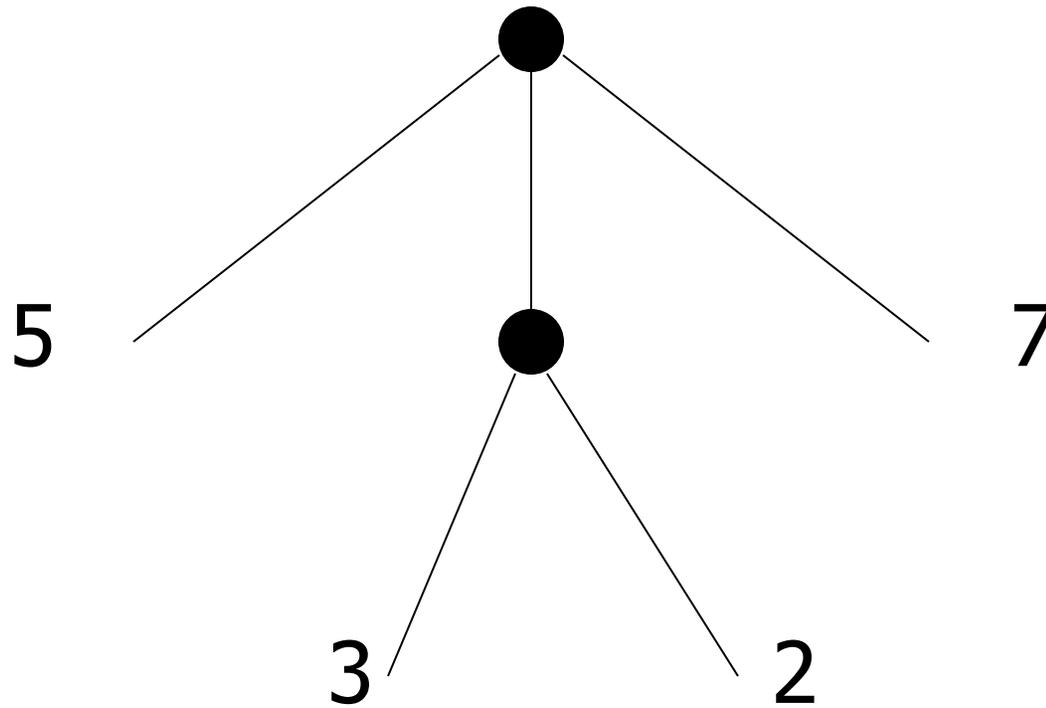
Mutually Recursive Types - Values

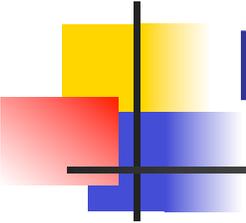




Mutually Recursive Types - Values

A more conventional picture



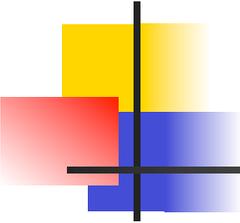


Mutually Recursive Functions

```
# 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);;
```

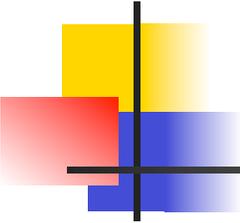
```
val fringe : 'a tree -> 'a list = <fun>
```

```
val list_fringe : 'a treeList -> 'a list = <fun>
```



Problem

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



Problem

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

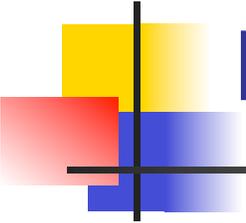
Define tree_size

```
let rec tree_size t =
```

```
    match t with TreeLeaf _ -> 1
```

```
    | TreeNode ts -> treeList_size ts
```

```
and
```



Problem

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

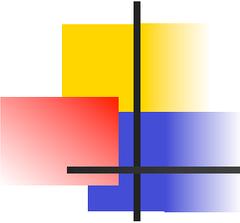
Define tree_size and treeList_size

```
let rec tree_size t =
```

```
    match t with TreeLeaf _ -> 1
```

```
    | TreeNode ts -> treeList_size ts
```

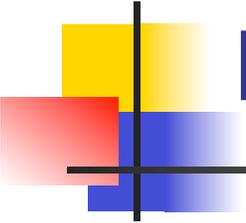
```
and
```



Problem

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

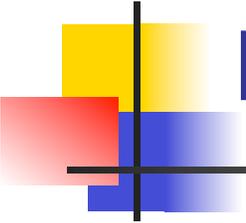
Define tree_size and treeList_size
let rec tree_size t =
  match t with TreeLeaf _ -> 1
  | TreeNode ts -> treeList_size ts
and treeList_size ts =
  match ts with Last t -> tree_size t
  | More t ts' -> tree_size t + treeList_size ts'
```



Mutually Recursive Functions

```
# fringe tree;;
```

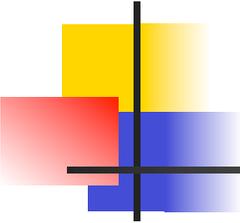
```
- : int list = [5; 3; 2; 7]
```



Nested Recursive Types

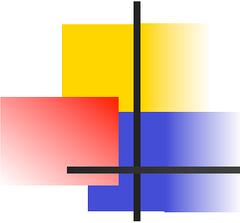
```
# type 'a labeled_tree =  
  TreeNode of ('a * 'a labeled_tree  
  list);;
```

```
type 'a labeled_tree = TreeNode of ('a  
  * 'a labeled_tree list)
```



Nested Recursive Type Values

```
# let ltree =  
  TreeNode(5,  
    [TreeNode (3, []);  
      TreeNode (2, [TreeNode (1, []);  
                          TreeNode (7, [])]);  
      TreeNode (5, [])]);;
```

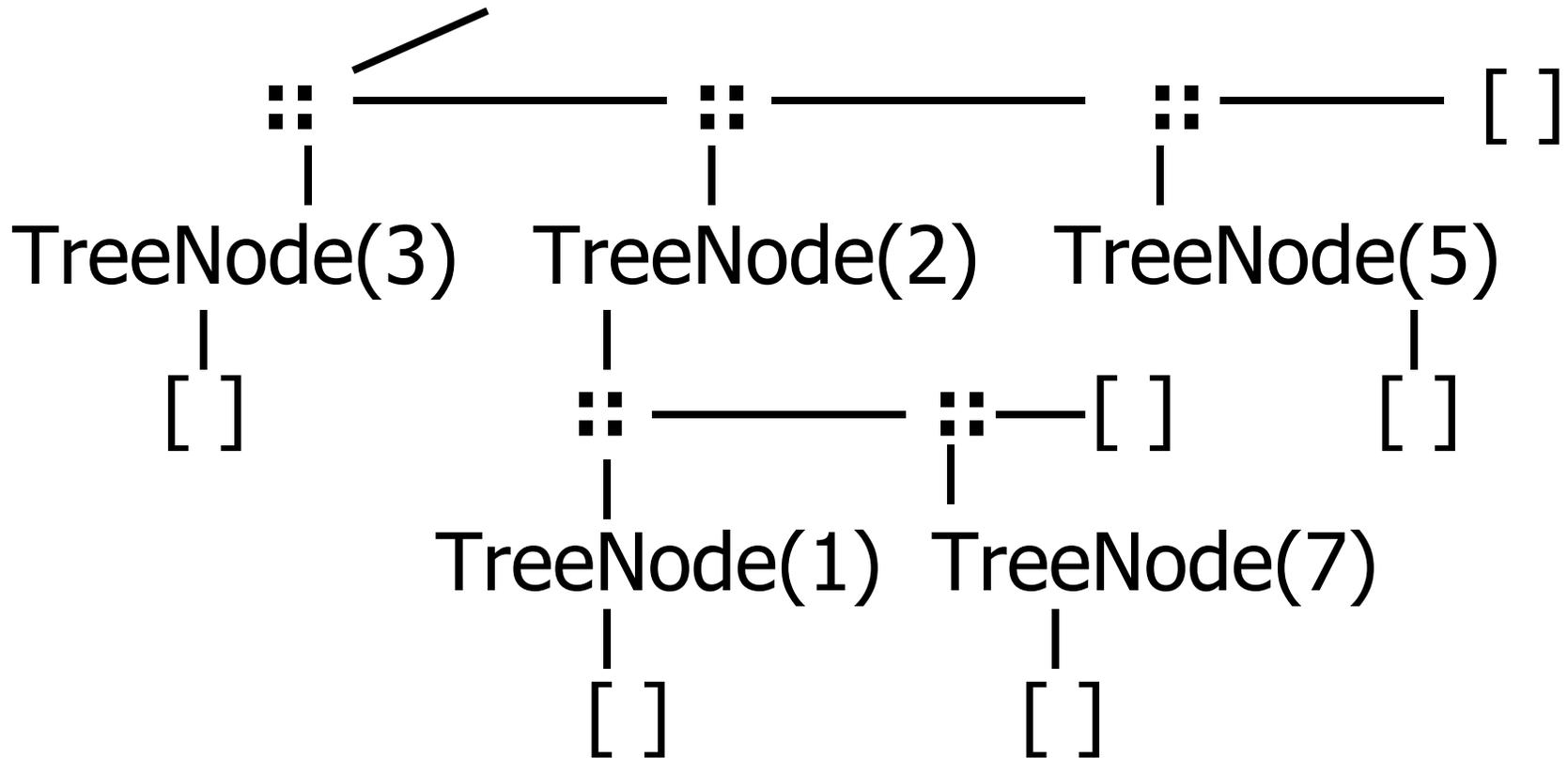


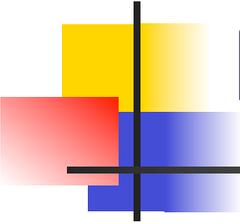
Nested Recursive Type Values

```
val ltree : int labeled_tree =  
  TreeNode  
    (5,  
      [TreeNode (3, []); TreeNode (2,  
        [TreeNode (1, []); TreeNode (7, [])]);  
        TreeNode (5, [])])
```

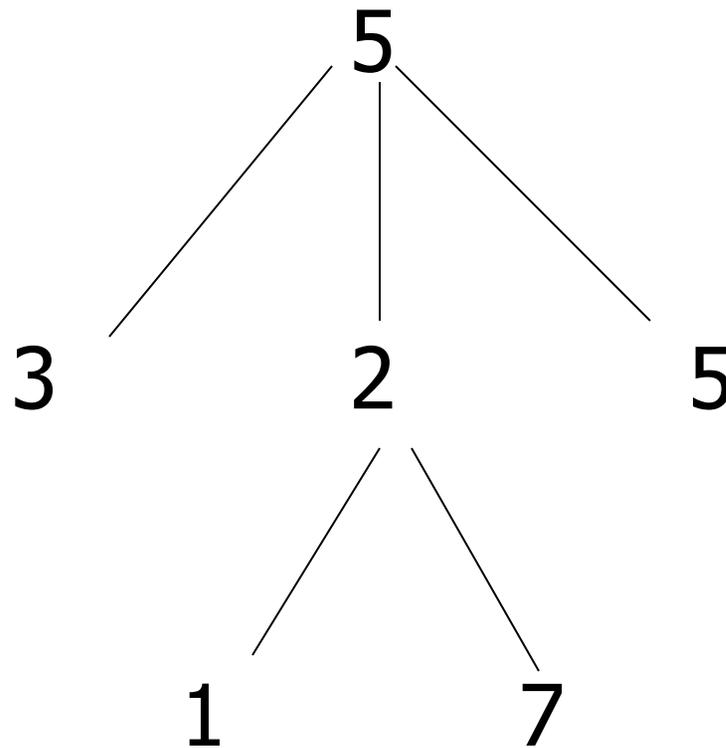
Nested Recursive Type Values

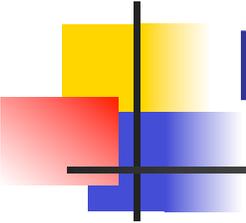
Ltree = TreeNode(5)





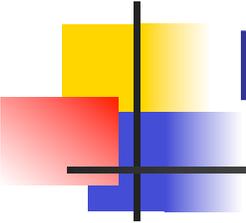
Nested Recursive Type Values





Mutually Recursive Functions

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



Mutually Recursive Functions

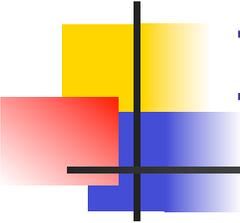
```
val flatten_tree : 'a labeled_tree -> 'a list =  
  <fun>
```

```
val flatten_tree_list : 'a labeled_tree list -> 'a  
  list = <fun>
```

```
# flatten_tree ltree;;
```

```
- : int list = [5; 3; 2; 1; 7; 5]
```

- Nested recursive types lead to mutually recursive functions



Infinite Recursive Values

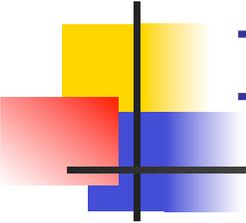
```
# let rec ones = 1::ones;;  
val ones : int list =  
  [1; 1; 1; 1; ...]  
# match ones with x::_ -> x;;
```

Characters 0-25:

Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:

```
[]
```

```
  match ones with x::_ -> x;;  
  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^  
- : int = 1
```

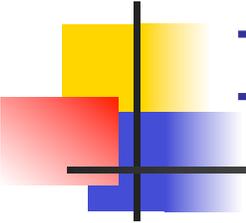


Infinite Recursive Values

```
# let rec lab_tree = TreeNode(2, tree_list)
  and tree_list = [lab_tree; lab_tree];;
```

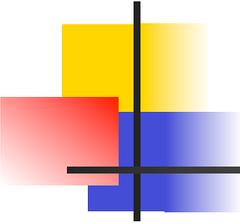
```
val lab_tree : int labeled_tree =
  TreeNode (2, [TreeNode(...); TreeNode(...)])
```

```
val tree_list : int labeled_tree list =
  [TreeNode (2, [TreeNode(...);
  TreeNode(...)]);
  TreeNode (2, [TreeNode(...);
  TreeNode(...)])]
```



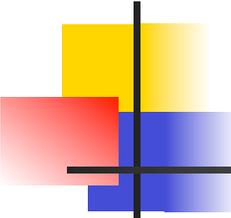
Infinite Recursive Values

```
# match lab_tree  
  with TreeNode (x, _) -> x;;  
- : int = 2
```



Records

- Records serve the same programming purpose as tuples
- Provide better documentation, more readable code
- Allow components to be accessed by label instead of position
 - Labels (aka *field names* must be unique)
 - Fields accessed by suffix dot notation



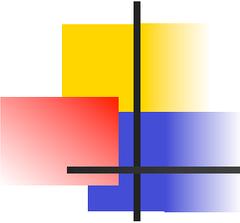
Record Types

- Record types must be declared before they can be used in OCaml

```
# type person = { name : string; ss : (int * int * int); age : int };;
```

```
type person = { name : string; ss : int * int * int; age : int; }
```

- person is the type being introduced
- name, ss and age are the labels, or fields



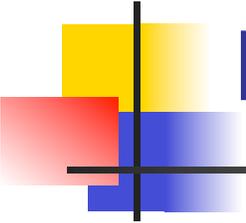
Record Values

- Records built with labels; order does not matter

```
# let teacher = {name = "Elsa L. Gunter";  
  age = 102; ss = (119,73,6244)};;
```

```
val teacher : person =
```

```
{name = "Elsa L. Gunter"; ss = (119, 73,  
  6244); age = 102}
```



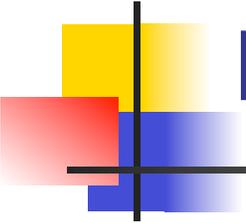
Record Pattern Matching

```
# let {name = elsa; age = age; ss =  
      (_,_,s3)} = teacher;;
```

```
val elsa : string = "Elsa L. Gunter"
```

```
val age : int = 102
```

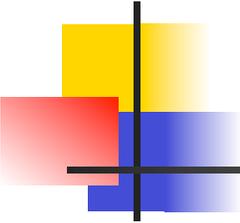
```
val s3 : int = 6244
```



Record Field Access

```
# let soc_sec = teacher.ss;;
```

```
val soc_sec : int * int * int = (119,  
73, 6244)
```



Record Values

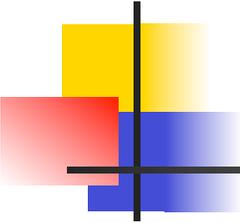
```
# let student = {ss=(325,40,1276);  
  name="Joseph Martins"; age=22};;
```

```
val student : person =
```

```
{name = "Joseph Martins"; ss = (325, 40,  
  1276); age = 22}
```

```
# student = teacher;;
```

```
- : bool = false
```



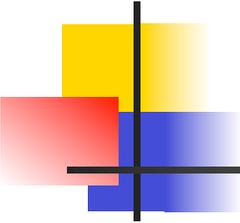
New Records from Old

```
# let birthday person = {person with age =  
    person.age + 1};;
```

```
val birthday : person -> person = <fun>
```

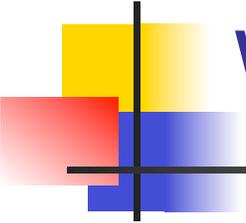
```
# birthday teacher;;
```

```
- : person = {name = "Elsa L. Gunter"; ss =  
    (119, 73, 6244); age = 103}
```



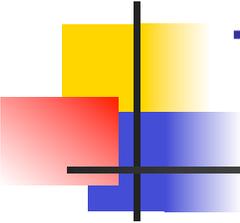
New Records from Old

```
# let new_id name soc_sec person =  
  {person with name = name; ss = soc_sec};;  
val new_id : string -> int * int * int -> person  
  -> person = <fun>  
# new_id "Giuseppe Martin" (523,04,6712)  
  student;;  
- : person = {name = "Giuseppe Martin"; ss  
  = (523, 4, 6712); age = 22}
```



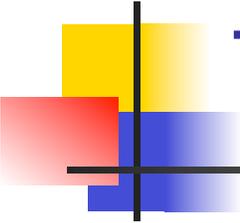
Why Data Types?

- Data types play a key role in:
 - *Data abstraction* in the design of programs
 - *Type checking* in the analysis of programs
 - *Compile-time code generation* in the translation and execution of programs



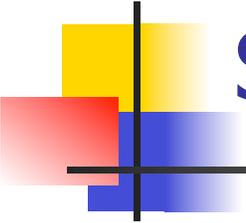
Terminology

- Type: A type t defines a set of possible data values
 - E.g. **short** in C is $\{x \mid 2^{15} - 1 \geq x \geq -2^{15}\}$
 - A value in this set is said to have type t
- Type system: rules of a language assigning types to expressions



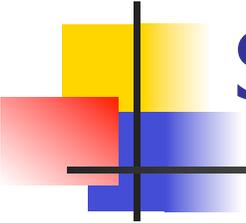
Types as Specifications

- Types describe properties
- Different type systems describe different properties, eg
 - Data is read-write versus read-only
 - Operation has authority to access data
 - Data came from “right” source
 - Operation might or could not raise an exception
- Common type systems focus on types describing same data layout and access methods



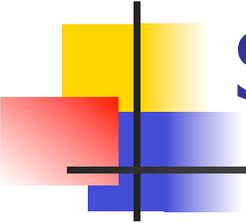
Sound Type System

- If an expression is assigned type t , and it evaluates to a value v , then v is in the set of values defined by t
- SML, OCAML, Scheme and Ada have sound type systems
- Most implementations of C and C++ do not



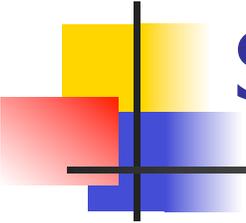
Strongly Typed Language

- When no application of an operator to arguments can lead to a run-time type error, language is *strongly typed*
 - Eg: `1 + 2.3;;`
- Depends on definition of “type error”



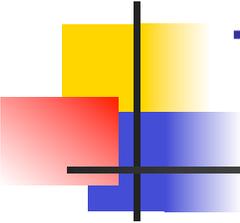
Strongly Typed Language

- 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



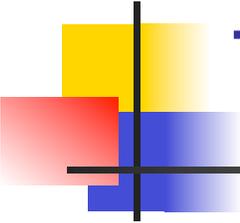
Static vs Dynamic Types

- *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



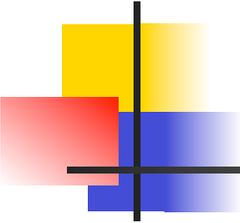
Type Checking

- When is $\text{op}(\text{arg1}, \dots, \text{argn})$ allowed?
- *Type checking* assures that 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



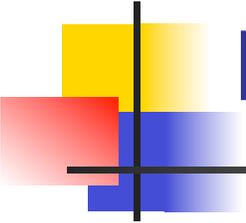
Type Checking

- Type checking may be done *statically* at compile time or *dynamically* at run time
- Dynamically typed (aka untyped) languages (eg LISP, Prolog) do only dynamic type checking
- Statically typed languages can do most type checking statically



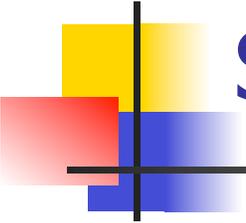
Dynamic Type Checking

- 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



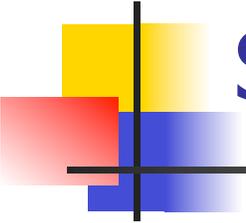
Dynamic Type Checking

- Data object must contain type information
- Errors aren't detected until violating application is executed (maybe years after the code was written)



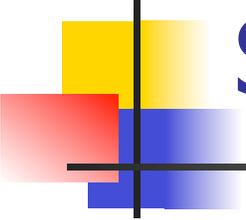
Static Type Checking

- Performed after parsing, before code generation
- Type of every variable and signature of every operator must be known at compile time



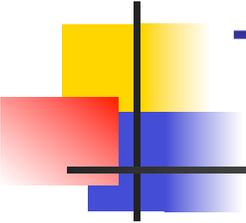
Static Type Checking

- 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
 - Eg: array bounds



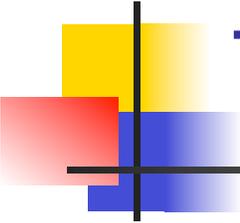
Static Type Checking

- Typically places restrictions on languages
 - Garbage collection
 - References instead of pointers
 - All variables initialized when created
 - Variable only used at one type
 - Union types allow for work-arounds, but effectively introduce dynamic type checks



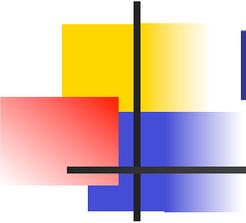
Type Declarations

- ***Type declarations***: explicit assignment of types to variables (signatures to functions) in the code of a program
 - 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

- *Type inference*: A program analysis to assign a type to an expression from the program context of the expression
 - 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

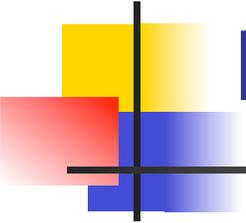


Format of Type Judgments

- A *type judgement* has the form

$$\Gamma \vdash \text{exp} : \tau$$

- Γ is a typing environment
 - Supplies the types of variables and functions
 - Γ is a list of the form $[x : \sigma , \dots]$
- exp is a program expression
- τ is a type to be assigned to exp
- \vdash pronounced “turnstile”, or “entails” (or “satisfies”)

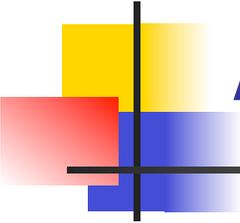


Format of Type Judgments

- A *type judgement* has the form

$$\Gamma \vdash \text{exp} : \tau$$

- Γ is a typing environment
 - Supplies the types of variables and functions
 - Γ is a list of the form $[x : \sigma , \dots]$
- exp is a program expression
- τ is a type to be assigned to exp
- \vdash pronounced “turnstile”, or “entails” (or “satisfies”)



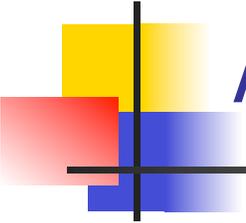
Axioms - Constants

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

$\overline{\quad}$
 $\vdash \text{true} : \text{bool}$

$\overline{\quad}$
 $\vdash \text{false} : \text{bool}$

- These rules are true with any typing environment
- n is a meta-variable

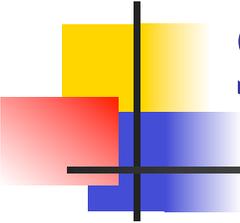


Axioms – Variables (Monomorphic Rule)

Notation: Let $\Gamma(x) = \sigma$ if $x : \sigma \in \Gamma$ and there is no $x : \tau$ to the left of $x : \sigma$ in Γ

Variable axiom:

$$\overline{\Gamma \vdash x : \sigma} \quad \text{if } \Gamma(x) = \sigma$$



Simple Rules - Arithmetic

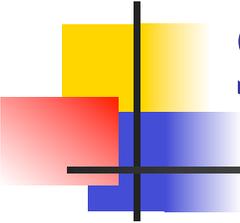
Primitive operators ($\oplus \in \{+, -, *, \dots\}$):

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau \quad (\oplus) : \tau \rightarrow \tau \rightarrow \tau}{\Gamma \vdash e_1 \oplus e_2 : \tau}$$

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

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 \sim e_2 : \text{bool}}$$

For the moment, think τ is int

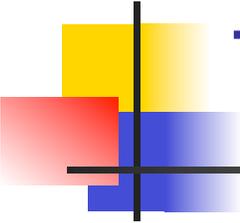


Simple Rules - Booleans

Connectives

$$\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \text{bool}}{\Gamma \vdash e_1 \ \&\& \ e_2 : \text{bool}}$$

$$\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \text{bool}}{\Gamma \vdash e_1 \ || \ e_2 : \text{bool}}$$

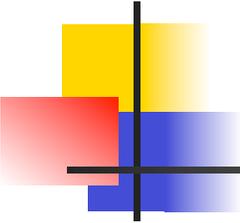


Type Variables in Rules

- If_then_else rule:

$$\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash (\text{if } e_1 \text{ then } e_2 \text{ else } e_3) : \tau}$$

- τ is a type variable (meta-variable)
- Can take any type at all
- All instances in a rule application must get same type
- Then branch, else branch and if_then_else must all have same type

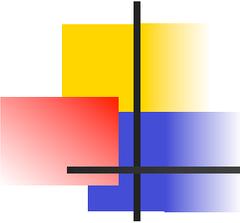


Function Application

- Application rule:

$$\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash (e_1 e_2) : \tau_2}$$

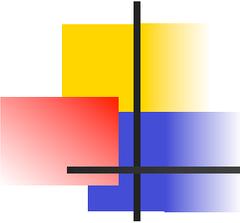
- If you have a function expression e_1 of type $\tau_1 \rightarrow \tau_2$ applied to an argument of type τ_1 , the resulting expression has type τ_2



Fun Rule

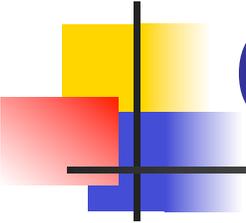
- Rules describe types, but also how the environment Γ may change
- Can only do what rule allows!
- fun rule:

$$\frac{[x : \tau_1] + \Gamma \vdash e : \tau_2}{\Gamma \vdash \text{fun } x \rightarrow e : \tau_1 \rightarrow \tau_2}$$



Fun Examples

$$\frac{[y : \text{int}] + \Gamma \vdash y + 3 : \text{int}}{\Gamma \vdash \text{fun } y \rightarrow y + 3 : \text{int} \rightarrow \text{int}}$$
$$\frac{[f : \text{int} \rightarrow \text{bool}] + \Gamma \vdash f \ 2 :: [\text{true}] : \text{bool list}}{\Gamma \vdash (\text{fun } f \rightarrow f \ 2 :: [\text{true}]) : (\text{int} \rightarrow \text{bool}) \rightarrow \text{bool list}}$$



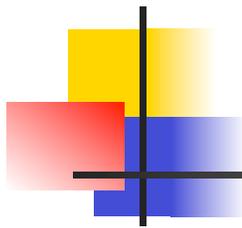
(Monomorphic) Let and Let Rec

- let rule:

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad [x : \tau_1] + \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (\text{let } x = e_1 \text{ in } e_2) : \tau_2}$$

- let rec rule:

$$\frac{[x : \tau_1] + \Gamma \vdash e_1 : \tau_1 \quad [x : \tau_1] + \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (\text{let rec } x = e_1 \text{ in } e_2) : \tau_2}$$

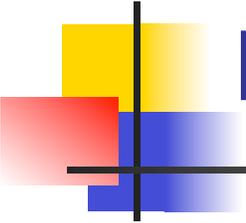


Example

- Which rule do we apply?

?

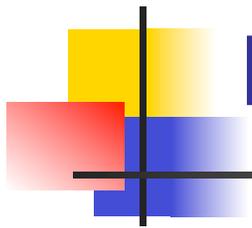
| - (let rec one = 1 :: one in
let x = 2 in
fun y -> (x :: y :: one)) : int → int list



Proof of 1

- Which rule?

$[one : int\ list] \vdash (1 :: one) : int\ list$



Proof of 1

- Application

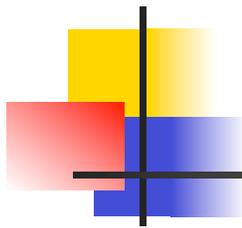
③

$$\frac{[one : int\ list] \ |- \ ((::) \ 1) : int\ list \rightarrow int\ list}{[one : int\ list] \ |- \ (1 \ :: \ one) : int\ list}$$

④

$$\frac{[one : int\ list] \ |- \ one : int\ list}{[one : int\ list] \ |- \ (1 \ :: \ one) : int\ list}$$

$$[one : int\ list] \ |- \ (1 \ :: \ one) : int\ list$$



Proof of 3

Constants Rule

Constants Rule

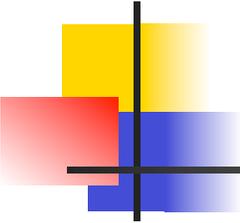
$[one : int\ list] \vdash$

$(::) : int \rightarrow int\ list \rightarrow int\ list$

$[one : int\ list] \vdash$

$1 : int$

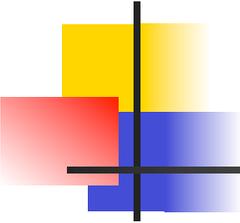
$[one : int\ list] \vdash ((::) 1) : int\ list \rightarrow int\ list$



Proof of 4

- Rule for variables

$$\overline{[one : \text{int list}] \vdash one:\text{int list}}$$



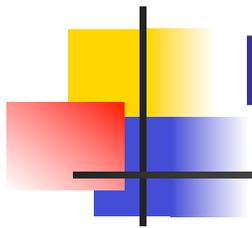
Proof of 2

- Constant

⑤ $[x:\text{int}; \text{one} : \text{int list}] \vdash$
 $\text{fun } y \rightarrow$
 $(x :: y :: \text{one}))$

$[\text{one} : \text{int list}] \vdash 2:\text{int} \quad : \text{int} \rightarrow \text{int list}$

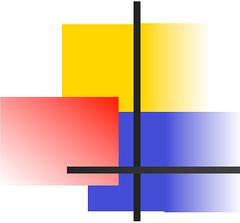
$[\text{one} : \text{int list}] \vdash (\text{let } x = 2 \text{ in}$
 $\text{fun } y \rightarrow (x :: y :: \text{one})) : \text{int} \rightarrow \text{int list}$



Proof of 5

?

$[x:\text{int}; \text{one} : \text{int list}] \vdash \text{fun } y \rightarrow (x :: y :: \text{one}))$
 $: \text{int} \rightarrow \text{int list}$

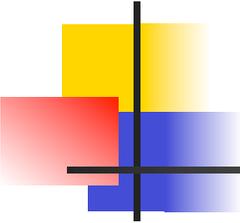


Proof of 5

?

$$[y:\text{int}; x:\text{int}; \text{one} : \text{int list}] \vdash (x :: y :: \text{one}) : \text{int list}$$

$$[x:\text{int}; \text{one} : \text{int list}] \vdash \text{fun } y \text{ -> } (x :: y :: \text{one})) \\ : \text{int} \rightarrow \text{int list}$$



Proof of 5

⑥

$[y:\text{int}; x:\text{int}; \text{one} : \text{int list}] \vdash [y:\text{int}; x:\text{int}; \text{one} : \text{int list}] \vdash$

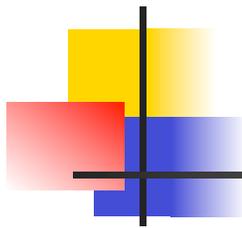
$((::) x):\text{int list} \rightarrow \text{int list}$

⑦

$(y :: \text{one}) : \text{int list}$

$[y:\text{int}; x:\text{int}; \text{one} : \text{int list}] \vdash (x :: y :: \text{one}) : \text{int list}$

$[x:\text{int}; \text{one} : \text{int list}] \vdash \text{fun } y \text{ -> } (x :: y :: \text{one})$
 $: \text{int} \rightarrow \text{int list}$



Proof of 6

Constant

Variable

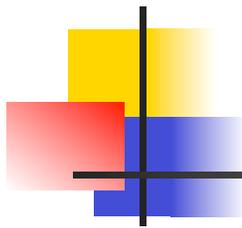
[...] |- (::)

: int → int list → int list

[...; x:int;...] |- x:int

[y:int; x:int; one : int list] |- ((::) x)

:int list → int list



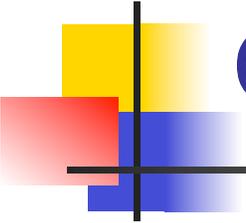
Proof of 7

Pf of 6 [y/x]

Variable

•
•
•

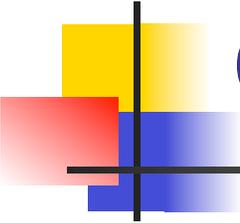
$$\frac{[y:\text{int}; \dots] \vdash ((::) y) \quad \vdash \text{int list} \rightarrow \text{int list}}{[y:\text{int}; x:\text{int}; \text{one} : \text{int list}] \vdash (y :: \text{one}) : \text{int list}}
 \quad
 \frac{[\dots; \text{one} : \text{int list}] \vdash \quad \vdash \text{one} : \text{int list}}{[\dots; \text{one} : \text{int list}] \vdash (y :: \text{one}) : \text{int list}}$$



Curry - Howard Isomorphism

- Type Systems are logics; logics are type systems
- Types are propositions; propositions are types
- Terms are proofs; proofs are terms

- Functions space arrow corresponds to implication; application corresponds to modus ponens



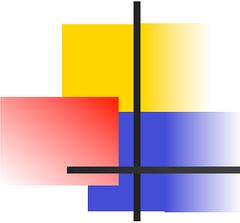
Curry - Howard Isomorphism

- Modus Ponens

$$\frac{A \Rightarrow B \quad A}{B}$$

- Application

$$\frac{\Gamma \vdash e_1 : \alpha \rightarrow \beta \quad \Gamma \vdash e_2 : \alpha}{\Gamma \vdash (e_1 e_2) : \beta}$$



Mia Copa

- The above system can't handle polymorphism as in OCAML
- No type variables in type language (only meta-variable in the logic)
- Would need:
 - Object level type variables and some kind of type quantification
 - **let** and **let rec** rules to introduce polymorphism
 - Explicit rule to eliminate (instantiate) polymorphism