Programming Languages and Compilers (CS 421)

Elsa L Gunter

2112 SC, UIUC

https://courses.engr.illinois.edu/cs421/sp2023

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

Problem

type int_Bin_Tree =Leaf of int

| Node of (int_Bin_Tree * int_Bin_Tree);;

- Write sum_tree : int_Bin_Tree -> int
- Adds all ints in tree

let rec sum_tree t =

Problem

```
type int_Bin_Tree =Leaf of int
| Node of (int_Bin_Tree * int_Bin_Tree);;
```

- Write sum_tree : int_Bin_Tree -> int
- Adds all ints in tree

```
let rec sum_tree t =
    match t with Leaf n -> n
    | Node(t1,t2) -> sum_tree t1 + sum_tree t2
```

Recursion over Recursive Data Types

```
# type exp = VarExp of string | ConstExp of const
  | BinOpAppExp of bin_op * exp * exp
  | FunExp of string * exp | AppExp of exp * exp
```

- How to count the number of variables in an exp?

Recursion over Recursive Data Types

```
# type exp = VarExp of string | ConstExp of const
  | BinOpAppExp of bin_op * exp * exp
  | FunExp of string * exp | AppExp of exp * exp
```

- How to count the number of variables in an exp?

```
# let rec varCnt exp =
  match exp with VarExp x ->
   | ConstExp c ->
   | BinOpAppExp (b, e1, e2) ->
   | FunExp (x,e) ->
   | AppExp (e1, e2) ->
```

Recursion over Recursive Data Types

```
# type exp = VarExp of string | ConstExp of const
  | BinOpAppExp of bin_op * exp * exp
  | FunExp of string * exp | AppExp of exp * exp
```

- How to count the number of variables in an exp?

```
# let rec varCnt exp =
  match exp with VarExp x -> 1
   | ConstExp c -> 0
   | BinOpAppExp (b, e1, e2) -> varCnt e1 + varCnt e2
   | FunExp (x,e) -> 1 + varCnt e
   | AppExp (e1, e2) -> varCnt e1 + varCnt e2
```

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)
```

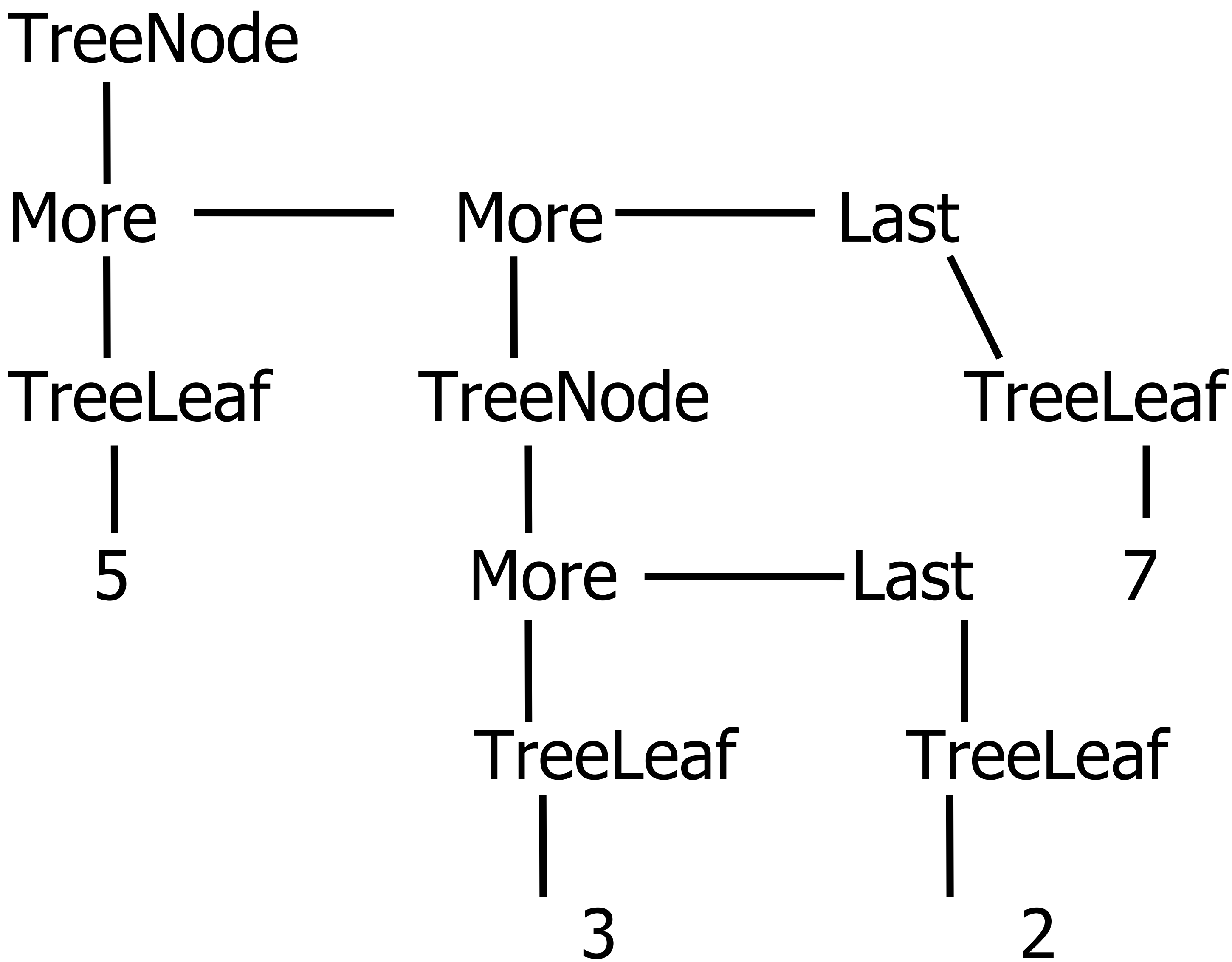
Mutually Recursive Types - Values

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

Mutually Recursive Types - Values

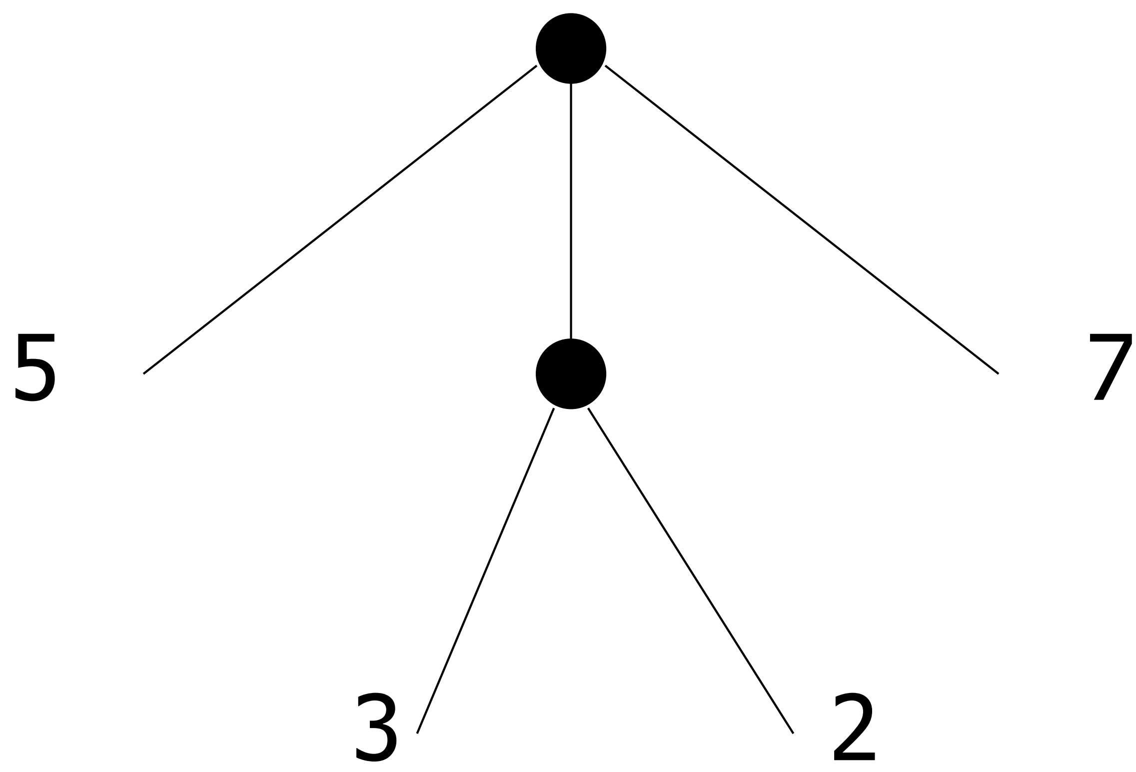
```
val tree : int tree =
 TreeNode
  (More
    (TreeLeaf 5,
     More
      (TreeNode (More (TreeLeaf 3, Last
  (TreeLeaf 2))), Last (TreeLeaf 7))))
```

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>
```

Mutually Recursive Functions

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

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 _ ->
      | TreeNode ts ->
```

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 + 1
```

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 + 1
and treeList_size ts =
```

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 + 1
and treeList_size ts =
    match ts with Last t ->
    | More (t, ts') ->
```

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 + 1
and treeList_size ts =
     match ts with Last t -> tree_size t
     | More (t, ts’) -> tree_size t + treeList_size ts’
```

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 + 1
and treeList_size ts =
    match ts with Last t -> tree_size t
    | More (t, ts') -> tree_size t + treeList_size ts'
```

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)
```

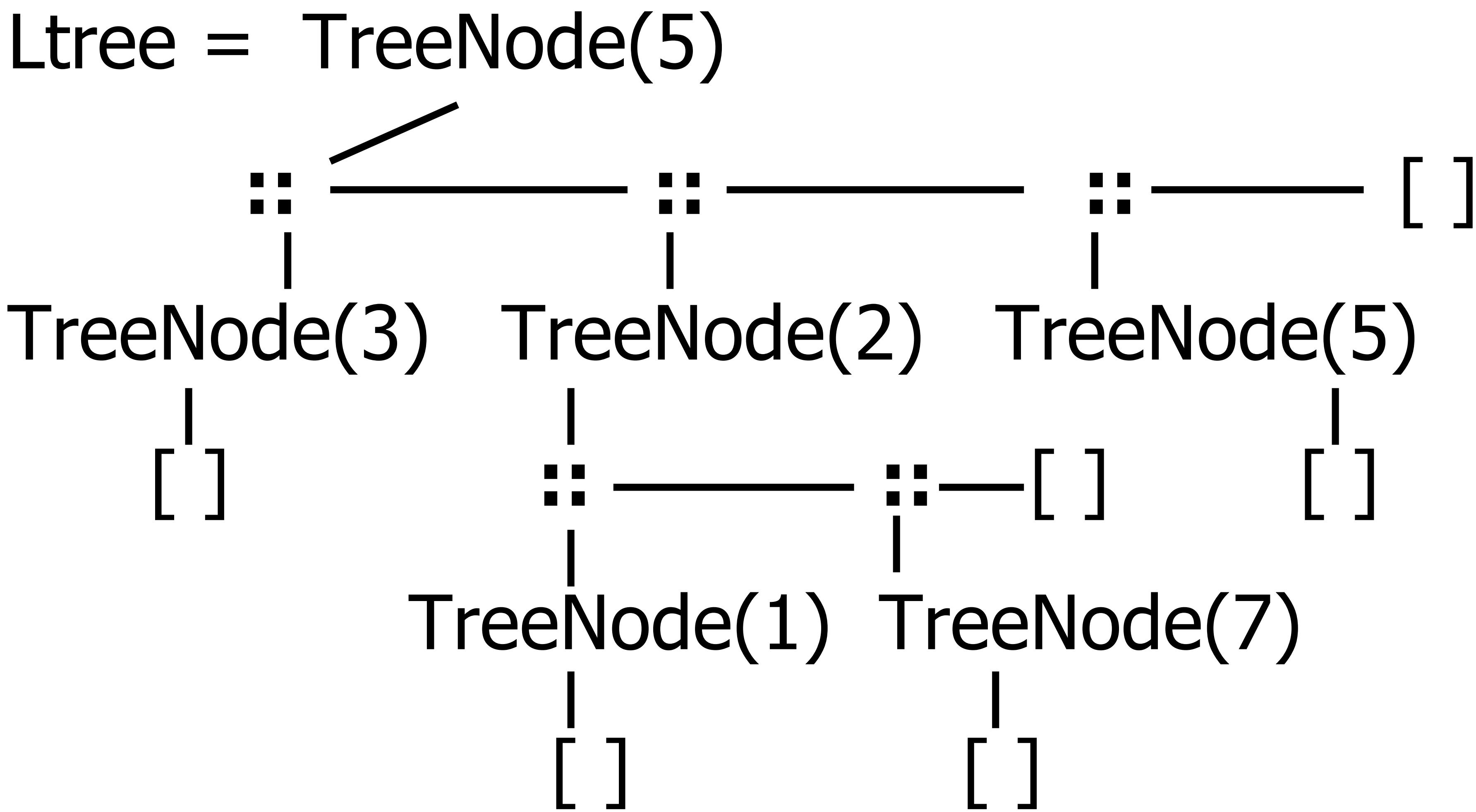
Nested Recursive Type Values

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

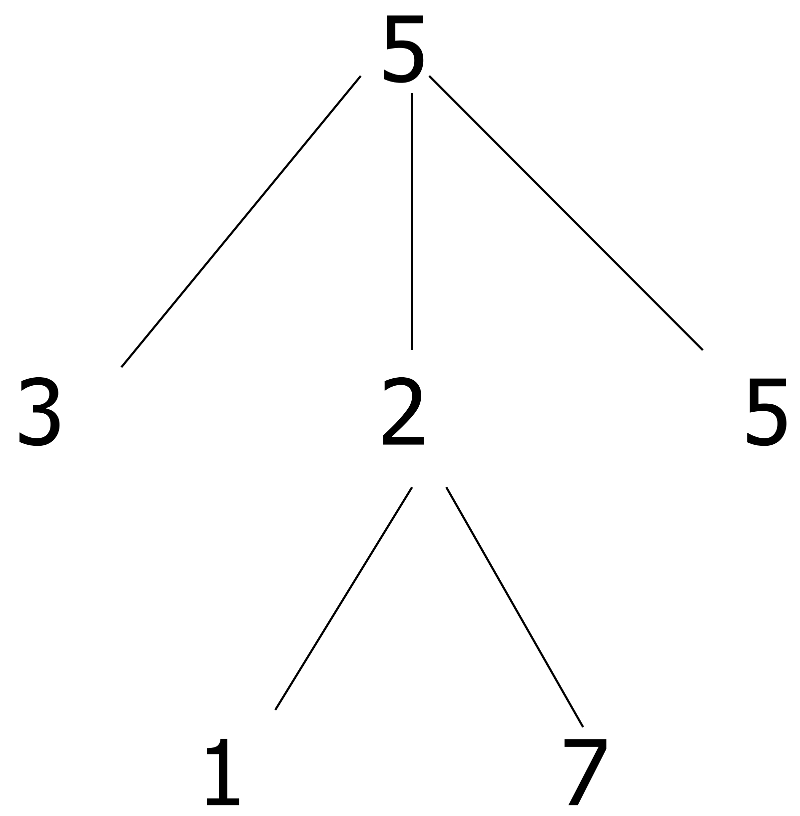
Nested Recursive Type Values

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

Nested Recursive Type Values



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

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
    - Data layout (how many words; which are data and which are pointers) dictated by type

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 op(arg1,…,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