

## **Announcements**

- **Final exam: May 10<sup>th</sup>, 4:00-7:00 pm (location TBA)**
- **Prolog programming assignment up (due May 3)**
- **No class on May 3 (what would have been the last day)**

1

## **Types**

- **What is a type?**
- **Type checking**
- **Type conversion**
- **Aggregates: strings, arrays, structures**
- **Enumeration types**
- **Subtypes**

2

# Types

- **What is a type?**
  - A set of values together with their meaningful operations
  - For example:
    - Integers: + - \* / mod < = ...
    - Booleans: and, or, not ...
    - Sets: union, intersection ...
    - Strings: concatenate, reverse ...
- **Why use types?**
  - to provide implicit context for many operations
  - to catch common (semantic) errors in programs

3

# Types

- **Three (complementary) points of view:**
  - **Denotational:** The meaning of an expression is a value from the *domain* that represents the expression's type.  
`x: int,     addone: int → int`
  - **Constructive:** Start with a small number of *primitive* types (e.g., integer, real, character, boolean), and form *composite* types by applying a type constructor (e.g., array, structure, set) to one or more simpler types.
  - **Abstract:** Specify a set of *operations* with a rigorous and mutually consistent semantics (e.g., interfaces)

4

# Type Systems

- **A type system is a set of rules for**
  - **constructing types (assuming constructive point of view)**
  - **determining or inferring the type of an expression, given the types of its components**
  - **determining type equivalence and allowing type conversion**
  - **determining type compatibility**
- **A type checker implements the type system**
  - **Goal: to determine, as early as possible, whether each function or operator in a program is supplied with the correct type of arguments**

5

# Type Systems

- **Programs may be typed**
  - **implicitly: types are determined by the way functions and variables are used in the program**
  - **explicitly: types are declared by the programmer**
- **Type checking may be performed**
  - **statically (at compile time): compute a type for every expression and check validity as program is compiled.**
  - **dynamically (at run time): store a type tag with every variable and check validity as program is executed.**
- **But there are mixtures and gradations of each.**

6

## Types of Expressions

- If  $f$  has type  $S \rightarrow T$  and  $x$  has type  $S$ , then  $f(x)$  has type  $T$ 
  - type of `3 div 2` is *int*
  - type of `round(3.5)` is *int*
- *Type error* - using wrongly typed operands in an operation
  - `round("hi")`
  - `3.5 div 2`
  - `"abc" + 3`

7

## Type Safety

- A *type safe* program executes on all inputs without type errors
    - Goal of type checking is to ensure type safety
    - Type safe does not mean no errors are present

```
if (n>0) then {
    y := "ab";
    if n<0 then y := y-5;
}
```
- Note that `x := y-5` is never executed so program is *type safe* (but contains an error).

8

# Strong Typing

- ***Strongly typed PL*** By definition, PL requires all programs to be type checkable
- ***Statically strongly typed PL*** - compiler allows only programs that can be type checked fully at compile-time
  - Algol68, ML
- ***Dynamically strongly typed PL*** -Operations include code to check runtime types of operands, if type cannot be determined at compile-time
  - Pascal, Java

9

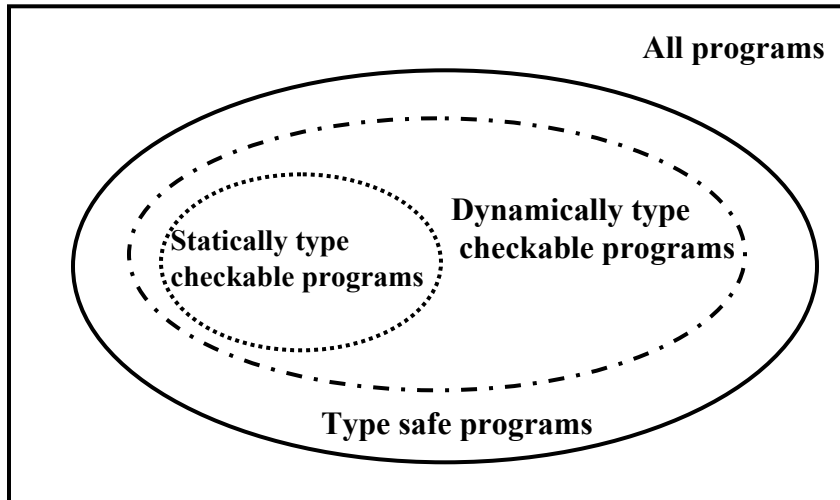
# Type Checking

- Kind of typing used is orthogonal to when complete type checking can be accomplished.

	Static	Dynamic
Implicit	ML	Scheme
Explicit	Ada	Pascal C

10

# Hierarchy of Programs



11

# Type Conversion

- **Implicit conversion: *coercion***
  - In C, mixed mode numerical operations

```
double d,e;  
e = d+2;    // 2 coerced to 2.0
```
  - Usually can use *widening* or conversion without loss of precision
    - integer → double, float → double
    - But double → int may lose precision and therefore cannot be implicitly coerced
  - Cannot coerce user-defined types or structures

12

# Type Conversion

- **Explicit conversion**
  - In Pascal, can explicitly convert types which may lose precision (*narrowing*)
    - `round(s)` `real` → `int` by rounding
    - `trunc(s)` `real` → `int` by truncating
  - In C, casting sometimes is explicit conversion
    - `(double)n` where `n` is declared as an `int`
    - `header *s;`  
`char *p = (char *)s;`  
Forces `s` to be considered as pointing to a `char` for purposes of pointer arithmetic

13

# Overloading Operators

- **Primitive type of *polymorphism***
  - When an operator allows operands of more than one type, in different contexts
- **Examples**
  - **Addition:** `2+3` is `5`, versus concatenation:  
`"abc" + "def" → "abcdef"`
  - **Comparison operator** used for two different types: `2 == 3` versus `"abc" == "def"`
  - **Integer addition:** `1+2` or **real addition:** `1.0+2.0`

14

## Definition of Arrays

- **Homogeneous, indexed collection of values**
- **Access to individual elements through subscript**
- **Choices made by a PL designer**
  - Subscript syntax
  - Subscript type, element type
  - When to set bounds, compile-time or runtime?
  - How to initialize?
  - What built-in operations allowed?

15

## Array Type

- **What is part of the array type?**
  - Size?
  - Bounds?
    - Pascal: bounds are part of type
    - C: bounds are not part of type
    - Must be fixed at compile-time in Pascal but can be set at runtime in C and Fortran
  - Dimension? always part of the type
- **Choice has ramifications on kind of type checking needed**

16