Programming Language Concepts
Data Types

Janyl Jumadinova
9 February, 2017
Examples of Data Types
Examples of Data Types

- Primitive data types
- Reference data types
- ADT (abstract data types)
What are types good for?
- implicit context
- checking
- make sure that certain meaningless operations do not occur
What are types good for?
- implicit context
- checking
- make sure that certain meaningless operations do not occur

Type checking cannot prevent all meaningless operations

It catches enough of them to be useful
Data Types

- What are types good for?
  - implicit context
  - checking
  - make sure that certain meaningless operations do not occur
- *Type checking* cannot prevent all meaningless operations
- It catches enough of them to be useful
- *Polymorphism* results when the compiler finds that it doesn’t need to know certain things
What Does “Implicit Context” Mean?

When we see a statement such as: \( x = y + z \);
are we:

- adding two int values, storing in an int?
- adding two int values, storing in a double?
- concatenating a string and an int, storing in a string
- adding an int and a double, storing in a double?
What Does “Implicit Context” Mean?

- If we were writing machine code, WE WOULD HAVE TO SPECIFY THIS (e.g., we would have to explicitly convert int to double before adding to a double or storing as a double; we would have to reserve space for the new string; etc.).
- Type information gives the compiler or interpreter a context that enables it to figure this out.
What is Polymorphism?

- In this context we are concerned with situations when the same variable can refer, at different times, to values of different types.
- The most familiar example to Java programmers occurs in subclasses.
Polymorphism

```java
public class A {
    int x;
    ...
}
public class B extends A {
    String x;
    ...
}

A first = new A();
B second = new B();
A third = new B();

What is the type of "third.x"?
int or String?
In Java, a subclass cannot override an instance variable of the parent class; however, it can “shadow it”. On the other hand, methods CAN be overridden.
STRONG TYPING has become a popular buzz-word
– like structured programming
– informally, it means that the language prevents you from applying an operation to data on which it is not appropriate
STRONG TYPING has become a popular buzz-word
– like structured programming
– informally, it means that the language prevents you from applying an operation to data on which it is not appropriate

STATIC TYPING
means that the compiler can do all the checking at compile time
Type Systems: Examples

- Common Lisp is strongly typed, but not statically typed
- Ada is statically typed
- Pascal is almost statically typed
- Java is strongly typed, with a non-trivial mix of things that can be checked statically and things that have to be checked dynamically
Type Systems: Common Terms

- **Discrete types** - countable
  - integer, boolean, char, enumeration, subrange
Type Systems: Common Terms

- **Discrete types** - countable
  - integer, boolean, char, enumeration, subrange

- **Scalar types** - one-dimensional
Type Systems: Common Terms

- **Discrete types** - countable
  - integer, boolean, char, enumeration, subrange
- **Scalar types** - one-dimensional
- **Composite types**
  - records (unions), arrays, sets, pointers, lists, files
Composite types

- Several values of varying types under a common name.
- Made somewhat obsolete by classes and instances in object-oriented programming.
Composite types

- Several values of varying types under a common name.
- Made somewhat obsolete by classes and instances in object-oriented programming.

```c
struct rec { /* here we declare the type */
    int i; double x; char s[10];
};
struct rec a,b,c; /* declare variables */
a.i = 10; b.x = 4.14
```
Composite types: Unions

Several values of varying types under a common name and sharing the same memory.

```c
union share { /* here we declare the type */
    int i; double x; char s[10];
};
union share a,b,c; /* declare variables */
a.i = 10; /* this changes a.x and a.s also */
```
Composite types: Enumerated types

Symbolic names (actual underlying values not important).

```c
enum weekday {mon, tue, wed, thu, fri, sat, sun};
enum weekday d;
d = mon;
if (d < fri) ...
```
Composite types: Subranges

- Lets you put bounds on the allowed values for certain variables.
- E.g., if we really want an integer variable to hold only values between 1 and 31, we can create a subrange type. (Pascal and Ada both have this.)

Example (Ada):

```ada
subtype Day is Integer range 1..31;
...
d: Day; -- d may hold only values 1,...,31
```
A collection of features is orthogonal if there are no restrictions on the ways in which the features can be combined.

ORTHOGONALITY is a useful goal in the design of a language, particularly its type system.
Orthogonality Example

Pascal is more orthogonal than Fortran, (because it allows arrays of anything, for instance), but it does not permit variant records as arbitrary fields of other records (for instance).
Orthogonality Example

Pascal is more orthogonal than Fortran, (because it allows arrays of anything, for instance), but it does not permit variant records as arbitrary fields of other records (for instance).

Orthogonality is nice primarily because it makes a language easy to understand, easy to use, and easy to reason about.
A TYPE SYSTEM has rules for:

- **type equivalence** (when are the types of two values the same?)
Type Checking

A TYPE SYSTEM has rules for:

- **type equivalence** (when are the types of two values the same?)
- **type compatibility** (when can a value of type A be used in a context that expects type B?)
- **type inference** (what is the type of an expression, given the types of the operands?)