Why types?

- **Untyped languages**
  - The programmer sees just words of memory
  - Drawbacks:
    - Need to remember which words are integers, floats, and pointers
    - Need to pick the correct operation
    - …and if we mess up it can have disasterous, undefined consequences
  - Most assembly languages are untyped
  - Example: x86 machine code
    - opcode MUL: unsigned multiplication
    - opcode IMUL: signed integer multiplication
    - opcode FMUL: multiply floating point
    - opcode FIMUL: multiply integer
    - Need to load into proper register to signify size (byte, word, double)
- **Type safety**
  - A program is type safe if it is guaranteed to have no type errors, i.e., it is guaranteed that its operations always apply to operands of the correct type.
  - A language is type safe if any program written in the language is type safe.

Benefits of types

- **Detect some programmer errors**
  - Trying to multiply a string and a number
  - Trying to add a character and a number
  - Trying to treat an integer as a Customer object
- **Serve as documentation**
  - Specify the interface to a reusable module of code
- **Hides the underlying representation**
  - Hides the details of a machine
  - Hides the details of a module implementation
- **More convenient notation**
  - Can multiply numbers of any kind with the same "*" operator
  - Can print or serialize objects of any kind, without saying what it is

What is a type?

- **Set of values**
- **Allowed operations**

Elementary types and user-defined types

- **Elementary types**
  - Built into the language
  - One step above the machine
    - Abstract away from the details of the machine
      - Example: A Java int is a 32-bit, two-complement integer
      - Example: A Common Lisp integer is a infinite-size, two-complement integer

- **User-defined types (composite types)**
  - Defined using constructors
  - Arrays, records, sets, variant records, pointers

Type systems

- **What is a type system?**
  - The set of rules used by a language to structure and organize its collection of types.
    - 1. Elementary types and constructors for new types
    - 2. Rules for determining types of expressions
    - 3. Rules for type equivalence and conversion

Dimensions of the design space for type systems

- **Binding time: static vs. dynamic**
  - Static: The type of every value can be determined at compile time
    - (For convenience, we include link time in compile time)
    - Explicit typing: Programmer supplies types via type declarations for each variable.
      - Example: Epsilon, C, Java
    - Type inference: The compiler implies the most general types
      - Example: ML, Haskell
  - Dynamic: The type of some values must be checked at run time
    - Even in a dynamically typed language, many types can be determined at compile time by type inference.
      - Thus, performance hit from dynamic type checking need not be large.
      - Programmer may supply optional type declartions in inner loops of performance-critical code.
      - Example: The CMU Common Lisp compiler
    - Examples: APL, SNOBOL, Lisp, most scripting languages
- **Strong vs. weak typing**
  - Strong typing
    - A type system is strong if it guarantees type safety.
    - A language with a strong type system is a strongly typed language.
  - Is a statically typed language necessarily strongly typed?
  - Is a strongly typed language necessarily statically typed?
- **Type compatibility/type equivalence**
  - Name compatibility
    - Definition: Two types T1 and T2 are name compatible if T1 = T2.
    - Subtypes (Pascal/Ada):
      - Different subtypes of a given type are considered to be compatible among themselves and with the supertype.
      - Example: type age = 0 .. maxint;
      - Ada subtypes do not define a new type: All values of all subtypes of INTEGER are of type INTEGER.
  - Structural compatibility

            - Definition: Two types T1 and T2 are structurally compatible if
                - They are name compatible, or
                - They are defined by applying the same type constructor to structurally compatible types.
            - Example 1: C
                - typedef int customer_id;
                - typedef int car_id;
                - Can pass a customer_id wherever a product_id is required.
            - Example 2: C
                - typedef struct { char *name, int age } customer;
                - typedef struct { char *make, int price } car;
                - Cannot pass a customer struct where a product struct is required.
            - Example 2: Haskell

```
type Customer = (String, Int)
type Car = (String, Int)

beater :: Car
beater = ("Honda", 3000)

make :: Car -> String
make (make, price) = make

deadbeat :: Customer
deadbeat = ("John Doe", 42)

name :: Customer -> String
name (name, age) = name

name(deadbeat) = "John Doe"
name(beater) = "Honda"
```

                - Can pass a Car wherever a Customer is required.
    - **Type conversion**
        - Coercions: Automatic conversions
            - float f = 3 + 4.0;
            - Ada does not have automatic conversions
                - i := INTEGER(4.0);
            - Javascript converts pretty much everything
                - "4" + 4 = 8
        - Casting: explicit conversions
            - int i = (int)4.0;
            - Going "around" the type system (untyped semantics)
    - **Monomorphic types vs. polymorphic types**
        - Simple, strong type system:
            - Every constant, variable, and routine has a declared type.

- Every operation requires an operand of that exact type.
- Such a system is called monomorphic (Greek: "single shape")
- Every object belongs to one and only one type
- Polymorphism: A value has more than one type
  - Example:
    - Integer is also Number; Float is also Number
    - Can use Integers and Floats wherever Numbers are required
- Polymorphic features in most or all languages
  - Type compatibility and coercion move us away from string monomorphism
- Classification of polymorphic features:
  - **Polymorphism**
    - **Universal**
      - **Parametric**
        - Generic functions
          - Can be applied to values of any type
          - Function that reverses a list can work on lists of any type.
            ```
            myreverse :: [a] -> [a]
            myreverse = foldl (flip (:)) []
            ```
          - Because the >= works on any ordinal type, a function that computes the max of two arguments can work on arguments of any ordinal type.
            ```
            mymax :: Ord t => (t, t) -> t
            mymax (a, b)
                | a >= b = a
                | otherwise = b
            ```
          - Ada and C++ fake this with templates, but it's not the same; should be considered ad-hoc polymorphism.
      - **Inclusion**
        - Subtyping
        - Example: Java
          - Data structures like Vector work on any subtype of Object
          - They are, however, treated like Objects, and need to be casted back to their subtype when retrieved from the Vector.
    - **Ad hoc**
      - **Overloading**
        - The + operator in C can be used with integers or floats.
        - Purely a syntactic convenience: Bound to "int+" or "float+" depending on the context.
      - **Coercion**
        - 3 + 4.0 in C
        - 3 coerced into 3.0 before the appropriate overloaded + operator is applied.

Reading assignment for next time

- **Rest of Chapter 3**