The Joy of Types

- All programming information is ultimately represented and manipulated as bit sequences…however, bit sequences aren't quite so natural for human beings, so we have the notion of *types*
- Types provide context and limitations for what can or cannot be done to the objects (in the generic sense) in a program
  - Is "a + b" floating-point or integer addition? The *types* of "a" and "b" help answer this question
  - Does it make sense to get the factorial of a file? Or to perform trigonometric functions on a collection? Types help to filter these out too

Overview of Type Concepts

- *Type systems* — definition and classification of types
- *Type checking* — rules that determine:
  - Type equivalence
  - Conversion and casts
  - Compatibility and coercion
  - Type inference
- *Type catalog* — records/structures, variants/unions, arrays, strings, sets, pointers/recursive types, lists, files and input/output

Type Systems

- *Type systems* consist of:
  - A mechanism for defining types and associating them with language constructs
  - Rules for *type equivalence*, *type compatibility*, and *type inference*
- Types are assigned to: anything that has a value, and anything that can refer to something that has a value
- *Type equivalence*: when do values have the same type?
- *Type compatibility*: under what context(s) can values of given types be used?
- *Type inference*: given any expression, what is its type?

- *Expressions vs. objects* — the type of an expression is not necessarily the type of the object to which it refers
- Subroutines are a distinct type in languages where they are first- or second-class values
- *Type checking* enforces the language's type system rules
  - A *type clash* is a violation of type compatibility rules
  - A language is *strongly typed* if it prohibits, in a way that can be enforced, the application of operations on objects that are not meant to support them
  - A language is *statically typed* if it is: (1) strongly typed and (2) type checking can be performed at compile time
    - Seldom have absolute 100% static typing — generally, we mean "most of the time"
  - A language is *dynamically typed* if it delays type checking until run-time
    - Dynamic typing is a form of late binding — types are not bound to objects until virtually the moment that types become relevant
    - Dynamic scope is associated with dynamic typing — after all, how can one check types statically if one doesn't know what an identifier refers to at compile time
    - Polymorphism does not necessarily imply dynamic type checking: e.g., Java, Eiffel

Type Definition

- Type *declaration* vs. *definition*
  - *Declaration*: existence and scope of a type
  - *Definition*: description of a type
- Language design choice: separate or single construct for type declaration and definition?
- Three perspectives on types: *denotational*, *constructive*, and *abstraction-based*

- *Denotational* perspective: types as sets of values
  - A value is of a given type if it belongs to the set of values that defines that type
  - Corresponds to the mathematical notion of *domains*
- *Constructive* perspective: types as either a built-in, simple, or primitive type, or a composite of these types (possibly arbitrarily nested) — the Composite design pattern applied to types
- *Abstraction-based* perspective: types as interfaces — types define a set of operations that may be performed
- The reality is that we mix and match these perspectives as needed

"Primitive," "Simple," or "Built-In" Types

- "Close to the hardware" — not much distance from bit-level representations
- *Booleans* (a.k.a. logicals) represent *true* or *false*
  - Some languages like C, Perl, and JavaScript allow other expressions to evaluate as "true" or "false" — in fact C and Perl have no separate boolean type per se, while JavaScript does
  - Instead of *true* or *false*, the Icon language uses *success* and *failure* — more general, in a way

	Evaluates to False	Evaluates to True
C	zero	anything else
Perl	non-zero number, non-empty string, non-empty array, non-empty map	everything else
JavaScript	*undefined*, *null*, +0, –0, *NaN*, empty string, *false*	everything else

- Characters represent individual symbols with human-attached meaning
  - Traditionally single bytes (ASCII), now becoming two bytes (Unicode — '\u0000' to '\u007f' correspond to ASCII)
- Numbers include integers, reals (floating point)
  - Different "widths" may be supported (32-bit, 64-bit): traditionally implementation/platform dependent, except in Java — Java specifies precision
  - Floating point alternatives: *rationals* (numerator, denominator), *fixed point* (implied decimal)
  - Signed vs. unsigned (*cardinals* in Modula-2) are sometimes distinguished
- Integers, booleans, characters, enumerations, and subranges are *discrete*/*ordinal* types: countable domains, clear successor/predecessor relationships
- Discrete, rational, real, and complex types are *scalar* or *simple* types: directly represented values, neither references nor composite

Enumeration Types

- Invented by Wirth in Pascal; emphasizes readability

```
type weekday = (sun, mon, tue, wed, thu, fri, sat);
```

  - Defines an order; allows use in enumeration-controlled loops
  - May be used to index arrays in some languages

- In other languages (C, Java < 1.5), enumerations are syntactic labels for other types (integers, strings)

  - Allows cross-usage of "enumerations" and other literals (e.g., integers, strings)
  - "True" enumerations in Java < 1.5 were simulated through clever class definitions
  - Language-level enumeration available in Java ≥ 1.5, but internally implemented as a class that follows a specific design pattern (in other words, syntactic sugar)

```
public enum Coin { penny(1), nickel(5), dime(10), quarter(25); ... }
```

Subrange Types

- Contiguous subsets of a discrete base or parent type

```
type test_score = 0..100;
type workday = mon..fri;
```

- Ada distinguishes between *derived* and *constrained* types

  - Derived types are a new, distinct type — not interchangeable with their base types
  - Constrained types are interchangeable

- Subrange types help to clarify the intent of program code; a form of non-comment documentation

- Subrange types also facilitate automated range checking or storage optimization

Composite Types

- a.k.a. *constructed* types: described by combining one or more scalar or composite types
- *Records* are collections of fields of other types, introduced by Cobol
  - Equivalent to mathematical tuples; the type itself corresponds to the Cartesian product of the types of the fields
- *Variant records* represent overlapping fields — the final type is a *union* of its named fields
- *Arrays* map an *index* type to a *component* type
  - Arrays of characters form *strings*, frequently given special treatment in many languages

- *Sets* represent unordered collections of a base type; introduced in Pascal, and follows the mathematical notion of sets closely
- *Pointers* are references to an object in that pointer's base type; generally, a pointer is an *l-value*
  - Basis of recursive types: types *T* that contain references to other objects of type *T*
- *Lists* are ordered collections of a base type, generally defined by its head and its tail — the head is of the base type, and the tail is another list
- *Files* or *streams* are structurally like arrays, but integrate a notion of *current position* and display idiosyncrasies due to physical I/O-bound behavior
- …we'll examine these in detail later on

Orthogonality in Types

Analogous to orthogonality in expressions and statements: how can type constructs be mixed and matched with other type and language constructs?

- Languages may define an "empty" type for constructs used solely for their side effects: *void* (C), *unit* (ML)
- Ability to use any discrete type to index an array, and to use any other type for the array's components, instead of just integers and scalars respectively (Pascal, in contrast to pre-ForTran 90)

- Subroutines as first-class values (several languages)
  - Special case: arbitrary blocks of code as first-class values (Smalltalk, Perl, JavaScript, ML)
  - Functions as types are particularly evident in ML (semantics of "curried" functions)
- Expressing values of any type (simple, composite) as some literal (C/C++, Java, Perl, Ada, ML)
  - Anonymous classes in Java allow inclusion of behavior (code) in addition to state (value)