We divide the universe of values according to types; a type is:
- a set of values; and
- a collection of operations defined on those values.

In practice, important to know how values are represented and how operations are implemented on real hardware.

Examples:
- **Integers** (represented by machine integers) with the usual arithmetic operations (implemented by corresponding hardware instructions).
- **Booleans** (represented by machine bits or bytes) with operators `and`, `or`, `not` (implemented by hardware instructions or code sequences).
- **Arrays** (represented by contiguous blocks of machine addresses) with operations like fetch and update (implemented by address arithmetic and indirect addressing).
- **Strings** (represented how?) with operations like concatenation, substring extraction, etc. (implemented how?)

### Hardware Types

Machine language doesn't distinguish types; all values are just bit patterns until used. As such they can be loaded, stored, moved, etc. But certain operations are supported directly by hardware; the operands are thus implicitly typed.

Typical hardware types:
- **Integers** of various sizes, signedness, etc. with standard arithmetic operations.
- **Floating point** numbers of various sizes, with standard arithmetic ops.
- **Booleans** with conditional branch operations.
- **Pointers** to values stored in memory.
- **Instructions**, i.e., code, which can be executed.
- Many others are possible, e.g., binary coded decimal.

Details of behavior (e.g., numeric range) are machine-dependent, though often subject to standards (e.g., IEEE floating point, Unicode characters, etc.).

### Language Primitive Types

**Primitive types** of a language are those whose values cannot be further broken down by user-defined code; they can be managed only via operators built into the language.

Usually includes hardware types plus others that can easily mapped to a hardware type.

Example: **enumeration** types are usually mapped to integers.

Numeric types only **approximate** behavior of true numbers. Also, they often inherit machine-dependent aspects of machine types, causing serious portability problems.

Example: Integer arithmetic in most languages.

Partial counterexample: Numerics in Scheme.

---

CS558 Programming Languages
Winter 2010
Lecture 13
Composite types are built from existing types using type constructors. Typical composite types include records, unions, arrays, functions, etc.

Abstractly, such type constructors can be seen as mathematical operators on underlying sets of simpler values. A small number of set operators suffices to describe most useful type constructors:

- **Cartesian product** \((S_1 \times S_2)\)
  - records, tuples, C structs
- **Sum or (disjoint) union** \((S_1 \oplus S_2)\)
  - enumerations, Pascal variant records, C unions
- **Mapping** \((S_1 \rightarrow S_2)\)
  - arrays, association lists, functions

In addition, we often need a way to represent recursive structures such as lists and trees.

Concretely, each language defines the internal representation of values of the composite type, based on the type constructor and the types used in the construction. Example: The fields of a record might occupy successive memory addresses (perhaps with some alignment restrictions). The total size of the record is (roughly) the sum of the field sizes.

Often a range of representations are possible, from highly packed to highly indirected. There’s often a tradeoff between space and access time.

Example: Arrays of booleans can be efficiently packed using one bit per element, but this makes it more complicated to read or set an element.

An important part of a language specification is distinguishing valid from invalid programs. It is useful to define three classes of errors that make programs invalid. (Of course, even valid programs may behave differently than the programmer intended!)

**Static errors** are violations of the language specification that can be detected at compilation time (or, in an interpreter, before interpretation begins)

- Includes: lexical errors, syntactic errors (caught during parsing), type errors, etc.
- Compiler or interpreter issues an error pinpointing erroneous location in source program.
- Language semantics are usually defined only for programs that have no static errors.

Checked runtime errors are violations that the language implementation is required to detect and report at runtime, in a clean way.

- Examples in Java: division by zero, array bounds violations, dereferencing a null pointer.
- Depending on language, implementation may issue an error message and die, or raise an exception (which can be caught by the program).
- Language semantics must specify behavior precisely.

Unchecked runtime errors are violations that the implementation need not detect.

- Subsequent behavior of the computation is arbitrary. (Error is often not manifested until much later in execution.)
- Examples in C: division by zero, dereferencing a null pointer, array bounds violations.
- Language semantics probably don’t specify behavior.
- Java and Scheme have no such errors!
### Static Typechecking

HLL’s differ from machine language in that explicit types appear and type violations are ordinarily caught at some point.

**Static typechecking** is most common.
- Types are associated with identifiers (esp. variables, parameters, functions).
- Every use of an identifier can be checked for type-correctness at compile time.
- “Well-typed programs don’t go wrong.”
- Compiler can optimize representations of values used at runtime.

### Dynamic Typechecking

Dynamic typechecking occurs in Lisp, Scheme, Smalltalk, many scripting languages, etc.
- Types are attached to values (usually as explicit tags).
- The type associated with identifiers can vary.
- Correctness of operations can’t (in general) be checked until runtime.
- Type violations become checked runtime errors.
- Optimized representation hard.

### Static Type Systems

The main goal of a type system is to characterize programs that won’t “go wrong” at runtime.

Informally, we want to avoid programs that confuse types, e.g., by trying to add booleans to integers, or take the square root of a string.

Formally, we can give a set of **typing rules** (sometimes called as static semantics) from which we can derive **typing judgments** about program fragments. (This should sound familiar from axiomatic semantics.)

Each judgment has the form

\[ TE \vdash e : t \]

Intuitively this says that expression \( e \) has type \( t \), under the assumption that the type of each free variable in \( e \) is given by the type environment \( TE \).

The key point is that an expression is well-typed if-and-only-if we can derive a typing judgment for it.

### Typing Rules

Consider a simple imperative language, and suppose we have just two types, \( \text{Int} \) and \( \text{Bool} \).

We write \( TE(x) \) for the result of looking up \( x \) in \( TE \), and \( TE + \{ x \mapsto t \} \) for the type environment obtained from \( TE \) by extending it with a new binding from \( x \) to \( t \).

Here is a suitable set of typing rules:

\[
\begin{align*}
\frac{x \in \text{dom}(TE)}{TE \vdash x : TE(x)} & \quad \text{(Var)} \\
\frac{TE \vdash i : \text{Int}}{TE \vdash \{+ e_1 e_2\} : \text{Int}} & \quad \text{(Add)}
\end{align*}
\]
The typing rules are just a formal system in which judgments can be derived. How do we connect this system with our slogan that “well-typed programs don’t go wrong”?

We need to relate the static type system to our dynamic semantics for execution, which we have been giving via an interpreter.

Roughly speaking, we must show that when we interpret a well-typed program, no runtime error occurs. (More precisely, only certain runtime errors are ruled out, such as the ones resulting from a failed check that the operands to addition are numbers.)

In fact, we typically show something a bit stronger: if \( \text{T E} \vdash e : t \), and running the interpreter on \( e \) terminates, then the interpretation yields a result \( v \) of type \( t \).

In practice, soundness proofs of this kind are usually formalized with respect to a dynamic semantics that is also presented as a system of deductive rules.

**Static Typechecking**

We can turn the typing rules into a recursive typechecking algorithm. A typechecker is very similar to the evaluators we have already built:

- it is parameterized by a type environment;
- it dispatches according to the syntax of the expression being checked (note that there is exactly one rule for each form);
- it calls itself recursively on sub-expressions;
- it returns a type.

There are some differences, though. For example, a typechecker always examines both arms of a conditional (not just one). If we consider a language with functions, the typechecker processes the body of each function only once, no matter how many times the function is called.

Note that most languages require the types of function parameters and return values to be declared explicitly. The typechecker can use this declaration to check that applications of the function are correctly typed, and separately checks that the body of the function is correctly typed.