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?)
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 LISP.
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.
**Composite Type Representation**

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.
High-level languages differ from machine language in that explicit types appear and type violations are ordinarily caught at some point.

**Static typechecking** is most common: FORTRAN, Algol, Pascal, C/C++, Java, ML, etc.

- 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.” (If type system is **sound**; often not true.)
- Compiler can optimize generated code because it knows about value representations.
Dynamic typechecking occurs in Lisp, Scheme, Smalltalk, Python, many other scripting languages, etc.

- Types are attached to values (usually as explicit tags).
- The type associated with an identifier can vary.
- Correctness of operations can’t (in general) be checked until runtime.
- Type violations become checked runtime errors.
- Optimized representation hard.

**Dynamic Typing**
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!)

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.
Consider our usual simple imperative language (see Lecture 3), 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:

$$
\frac{TE(x) = t}{TE \vdash x : t} \quad \text{(Var)}
$$

$$
\frac{}{TE \vdash i : \text{Int}} \quad \text{(Int)}
$$

$$
\frac{TE \vdash e_1 : \text{Int} \quad TE \vdash e_2 : \text{Int}}{TE \vdash (\+ e_1 e_2) : \text{Int}} \quad \text{(Add)}
$$
Typing Rules (2)

\[
\begin{align*}
  & TE \vdash e_1 : \text{Int} \quad TE \vdash e_2 : \text{Int} \quad (\text{Leq}) \\
  \hline
  & TE \vdash (\leq e_1 e_2) : \text{Bool} \\
  & \quad \frac{TE \vdash e_1 : t_1 \quad TE + \{x \mapsto t_1\} \vdash e_2 : t_2}{TE \vdash (\text{local } x e_1 e_2) : t_2} \quad (\text{Local}) \\
  & \quad \frac{TE(x) = t \quad TE \vdash e : t}{TE \vdash (\text{:= } x e) : t} \quad (\text{Assgn}) \\
  & \quad \frac{TE \vdash e_1 : \text{Bool} \quad TE \vdash e_2 : t \quad TE \vdash e_3 : t}{TE \vdash (\text{if } e_1 e_2 e_3) : t} \quad (\text{If}) \\
  & \quad \frac{TE \vdash e_1 : \text{Bool} \quad TE \vdash e_2 : t}{TE \vdash (\text{while } e_1 e_2) : \text{Int}} \quad (\text{While})
\end{align*}
\]
The typing rules are just (another) 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”? 

First we need an auxiliary judgment system assigning types to values, written $\vdash v : t$.

For example, we would have $\vdash i : \text{Int}$ for every integer $i$, $\vdash \text{true} : \text{Bool}$, and $\vdash \text{false} : \text{Bool}$

We also add a special value error which does not belong to any type: $\not\vdash \text{error} : t$

We extend this notation to environments and stores, and write $\vdash E, S : TE$ iff $\text{dom}(E) = \text{dom}(TE)$ and $\vdash S(E(x)) : TE(x), \forall x \in \text{dom}(E)$. 
Recall our formal **dynamic semantics** for our language, defined using $\downarrow$ judgments. In our previous definition, expressions corresponding to runtime errors simply had no applicable rule (they were “stuck.”). Let’s change the system slightly by adding new rules so that all expressions corresponding to runtime errors evaluate to `error` instead.

Now, if everything has been defined correctly, we should be able to prove a **theorem** roughly like this:

If $TE \vdash e : t$ and $\models E, S : TE$ and $\langle e, E, S \rangle \downarrow \langle v, S' \rangle$ then $\models v : t$.

In other words, well-typed programs evaluate to values of the expected type; so in particular, they can’t evaluate to `error`, which belongs to no type.
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.
Static typechecking offers the great advantage of **catching errors early**, and generally supports more efficient execution.

Why ever consider dynamic typechecking?

- Simplicity. For short or simple programs, it’s nice to avoid the need for declaring the types of identifiers.

- Flexibility. Static typechecking is inherently more **conservative** about what programs it admits. For example, suppose function \( f \) happens to always return \( \text{false} \). Then

\[
(\text{if } f() \text{ then } "a" \text{ else } 2) + 2
\]

will never cause a runtime type error, but it will still be rejected by a static type system.

Perhaps more usefully, dynamic typing allows **container** data structures, to contain mixtures of values of arbitrary types, like this list:

\[
[2; \text{true}; 3.14]
\]
Some statically-typed languages, like OCaml, offer alternative ways to approach these goals, via type inference and polymorphic typing.

Type inference works like this:

- The types of identifiers are automatically inferred from the way they are used.
- The programmer is no longer required to declare the types of identifiers (although this is still permitted).
- Requires that the types of operators and literals are known.
(Assume just \texttt{int} and \texttt{bool} as base types.)

\begin{verbatim}
let f x = x + 2
  in f y
end
\end{verbatim}

The type of \texttt{x} must be \texttt{int} because it is used as an arg to +. So the type of \texttt{f} must be \texttt{int \rightarrow int}, and \texttt{y} must be an \texttt{int}.

\begin{verbatim}
let f x = [x]
  in f true
end
\end{verbatim}

Suppose \texttt{x} has some type \texttt{t}. Then the type of \texttt{f} must be \texttt{t \rightarrow t list}. Since \texttt{f} is applied to a \texttt{bool}, we must have \texttt{t = bool}.

(For the moment, we’re assuming the \texttt{f} must be given a unique \textbf{monomorphic} type; in real ML, this isn’t true...)
A harder example:

```plaintext
let f x = if x then p else q
in 1 + (f r)
end
```

Can only infer types by looking at both the function’s body and its applications.

In general, we can solve the inference task by extracting a collection of typing constraints from the program’s AST, and then finding a simultaneous solution for the constraints using unification.

Extract constraints that tell us how types must be related if we are to be able to find a typing derivation. Each node generates one or more constraints.
Inference for ML-like functions

We can rewrite the example slightly as

```plaintext
let f = fun x -> if x then p else q
in 1 + (f r)
end
```

We’ll need some extra type judgment rules:

\[
\begin{align*}
    & TE + \{x \mapsto t_1\} \vdash e : t_2 \\
    \Rightarrow \\
    & TE \vdash \text{fun } x \to e : t_1 \to t_2 \\
    \uparrow \\
    & (\text{Fn}) \\
\end{align*}
\]

\[
\begin{align*}
    & TE \vdash e_1 : t_1 \to t_2 \quad TE \vdash e_2 : t_1 \\
    \Rightarrow \\
    & TE \vdash e_1 e_2 : t_2 \\
    \uparrow \\
    & (\text{Appl})
\end{align*}
\]
### Solving Inference Constraints

<table>
<thead>
<tr>
<th>Node</th>
<th>Rule</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Let</td>
<td>( t_f = t_2 ) \hspace{1cm} t_1 = t_7</td>
</tr>
<tr>
<td>2</td>
<td>Fun</td>
<td>( t_2 = t_x \rightarrow t_3 )</td>
</tr>
<tr>
<td>3</td>
<td>If</td>
<td>( t_4 = \text{bool} ) \hspace{1cm} t_3 = t_5 = t_6</td>
</tr>
<tr>
<td>4</td>
<td>Var</td>
<td>( t_4 = t_x )</td>
</tr>
<tr>
<td>5</td>
<td>Var</td>
<td>( t_5 = t_p )</td>
</tr>
<tr>
<td>6</td>
<td>Var</td>
<td>( t_5 = t_q )</td>
</tr>
<tr>
<td>7</td>
<td>Add</td>
<td>( t_7 = t_8 = t_9 = \text{int} )</td>
</tr>
<tr>
<td>8</td>
<td>Int</td>
<td>( t_8 = \text{int} )</td>
</tr>
<tr>
<td>9</td>
<td>Appl</td>
<td>( t_{10} = t_{11} \rightarrow t_9 )</td>
</tr>
<tr>
<td>10</td>
<td>Var</td>
<td>( t_{10} = t_f )</td>
</tr>
<tr>
<td>11</td>
<td>Var</td>
<td>( t_{11} = t_r )</td>
</tr>
</tbody>
</table>

**Solution:**

\( t_1 = t_7 = t_8 = t_9 = t_3 = t_5 = t_p = t_6 = t_q = \text{int} \)

\( t_4 = t_x = t_{11} = t_r = \text{bool} \)

\( t_2 = t_f = t_{10} = \text{bool} \rightarrow \text{int} \)
Consider this variant program:

```plaintext
let f x = if x then p else false
in 1 + (f r)
end
```

Now the body of \( f \) return type \texttt{bool}, but it is used in a context expecting an \texttt{int}.

The corresponding extracted constraints will be \texttt{inconsistent}; no solution can be found. Can report this to the programmer.

But which is wrong, the definition of \( f \) or the use? Doesn’t really work to associate the error message with a single program point. (In general, may need to consider an arbitrarily long chain of program points.)
Consider

```ml
let head l = match l with x::xs -> x in
head [1;2;3]
```

By extracting the constraints as above, and solving, we will conclude that

head has type \texttt{int list} \to \texttt{int}.

It is also perfectly sensible to write:

```ml
let head l = match l with x::xs -> x in
head [true;false;true]
```

giving head the type \texttt{bool list} \to \texttt{bool}. Note that the definition of head hasn’t changed at all!

So reasonable to ask: why can’t we write something like:

```ml
let head l = match l with x::xs -> x in
  (head [true;false;true],
   head [1;2;3])
```

Can do this if we treat the type of head as \texttt{polymorphic}. 
By default OCaml infers the most polymorphic possible type for every function. In this case, it would give \texttt{head} the type \( \forall \alpha \, \text{list} \rightarrow \alpha \), where \( \alpha \) (pronounced “alpha”) is implicitly universally quantified. Each use of \texttt{head} occurs at a particular instance of \( \alpha \) (first at \texttt{bool}, then at \texttt{int}).

This is called \textbf{parametric polymorphism} because the function definition is (implicitly) parameterized by the instantiating type.

In this model, the \textit{behavior} of the polymorphic function is \textbf{independent} of the instantiating type. In fact, an Ocaml compiler typically generates just one piece of object code for each polymorphic function, shared by all instances. (More later.) An alternative is to generate type-specific versions of the code for each different instance.
Most languages provide some form of overloading, where the same symbol means different things depending on the types to which it is applied. E.g., overloading of arithmetic operators to work on either integers or reals is very common.

Aim is to do “what we expect;” rules can get quite complicated (especially when coercions are considered)!

Some languages (e.g., Ada, C++) support user-defined overloading, normally for user-defined types (e.g. complex numbers).

In conventional languages, overloading is resolved statically; that is, the compiler selects the appropriate version of the operator once and for all at compiler time. (Different from object-oriented dynamic overriding; more later.)

Overloading is sometimes called “ad-hoc polymorphism”. It is fundamentally different from parametric polymorphism, because the implementation of the overloaded operator changes according to the underlying types.
ML has a well-conceived set of basic type constructs. (OCaml adds some more controversial extensions.)

- **Primitives:** unit, int, float, char, string, exn, array, ...

- **Product (record or tuple) types** \((t_1 \times t_2)\):

  ```ml
type emp = string * int  (* tuple: unlabeled fields *)
  let x : emp = ("abc",3)
  type emp =
    {name: string; age: int}  (* record: labeled fields *)
  let x : emp = {name="abc"; age=3}
  ```

  Tuple (but not record) values may be written without declaring an explicit named type first.

- **Functions:** \(t_1 \rightarrow t_2\)

  All functions take just one argument; the effect of multi-argument functions can be obtained by passing a product or by Currying.
ML’s **type** mechanism can be used to define many different useful types.

- Each **type** declaration defines a new type and specifies its **data constructors** (which take 0 or more arguments).
- Value of the type are taken apart using **pattern matching** in a case statement or function declaration.

**Sums** \((t_1 \oplus t_2)\)

```ml
type temp = F of float
           | C of float
let boiling (t:temp) : bool = match t with
    | F r -> r >= 212.0
    | C r -> r >= 100.0
```

Can combine `match` into anonymous function definition, e.g.

```ml
let boiling = function
    | F r -> r >= 212.0
    | C r -> r >= 100.0
```
Recursive types

type inttree = Branch of inttree * inttree
  | Leaf of int
let rec sumleaves = function
  Leaf i -> i
  | Branch(l,r) -> (sumleaves l) + (sumleaves r)

Parameterized type constructors (polymorphic types)

type 'a bintree = Branch of 'a bintree * 'a bintree
  | Leaf of 'a
let rec depth = function
  Leaf _ -> 0
  | Branch(l,r) -> max (depth l) (depth r) + 1

Type 'a list is just a special case of a parameterized type constructor, with extra syntactic sugar for writing literals.)
Enumerations

type day =
    Mon | Tue | Wed | Thu | Fri | Sat | Sun
let weekday (d:day) : bool =
    match d with
    Sat -> false
| Sun -> false
| _ -> true

(Type bool is just a special case of an enumeration.)

Singleton types

type complexR = CR of float * float
type complexP = CP of float * float
let convert (CP (r,theta)) =
    CR(r *. (cos theta),r *. (sin theta))
let x : complexR = CR(1.0,-1.0)
... convert x ... (* STATIC TYPE ERROR ! *)
Ocaml also has type **abbreviations**, which are also introduced by type declarations. These simply serve to give convenient names to possibly lengthy types.

```ocaml
type t = int * bool (* note: no constructor name *)
let x : t = (2, true)
let f ((a:int),(b:bool)) = ...
... f x ...

(* TYPE-CHECKS FINE *)
```
Basic representation idea for user-defined datatypes: each value is represented boxed, more specifically as a two-element record, containing a tag field and a contents field (which may itself be a record).

Example: Trees

```ocaml
type tree = Leaf of int
  | Tree of tree * tree
```

Leaf(x)  
\[
\begin{array}{c}
0 \\
x
\end{array}
\]

Tree(y,z)  
\[
\begin{array}{c}
1 \\
y \\
z
\end{array}
\]
Tree(Leaf(11), Tree(Leaf(22), Leaf(33)))
The above scheme is not very efficient for important special classes of datatypes, so in practice certain optimizations are used.

- **Nullary** constructors (like enumeration values) are represented as unboxed small integers.

- List values are represented without internal indirections:

  
  \[ [11, 22, 33] \]

  Every value still occupies just one word (boxed or unboxed) This is an example of **uniform data representation**. ML implementations usually use this representation (although they are not required to). This makes it particularly easy to generate code for polymorphic functions.