Goals

- Introduce Data Types
Data Types

• Computers manipulate **sequences of bits**
• We manipulate **higher level data** (numbers, strings, etc.)
• **Data types** transform bits into higher level data
Data Types:

• Types provide **implicit context**
  
  • **Compilers can infer information**, so programmers write less code.
  
  • e.g., The expression `a+b` in Java may be adding two `integer`, two `floats` or two strings depending on **context**

• Types provide a set of **semantically valid operation**
  
  • Compilers can **detect semantic mistakes**
  
  • e.g., Python’s list support `append()` and `pop()`, but complex numbers do not
Type Systems

• A type system consists of
  1. A mechanism to define types and associate them with language constructs
  2. A set of rules for “type equivalence,” “type compatibility,” and “type inference.”
Type Systems

A type system consists of:

1. A mechanism to define types and associate them with language constructs.

Discuss these in detail.
Type Systems: Type Checking

- **Type Checking** is the process of ensuring that a program **obeys the language’s type compatibility rules**
  - Strongly typed.
  - Weakly typed.
Strongly Typed

• **Strongly typed** languages *always detect type errors*
  - All *expressions and objects* must have a type
  - All operations must be applied in *appropriate type contexts*

• **Statically typed** languages are *strongly typed* languages in which *all type checking occurs at compiled time*
Strongly Typed

- Strongly typed languages detect type errors:
  - All expressions and objects must have a type.
  - All operations must be applied in appropriate type contexts.

- Statically typed languages are strongly typed languages in which all type checking occurs at compiled time.

Even FUNCTIONS!
Weakly Typed

- In **weakly typed** languages “anything can go”
  - e.g., perl.
Typing and “Degree of Abstraction”

Degree of Typing

Instructions/Statement (Level of Abstraction)

Assembly

Tcl/Perl

System Prog.

Visual Basic

Scripting

C

C++

Java

None

Strong
Typing and “Degree of Abstraction”

- Assembly
- C
- C++
- Java
- Tcl/Perl
- Visual Basic

Instructions/Statement (Level of Abstraction)

Degree of Typing

- None
- Strong

Very strongly typed languages won’t allow “implicit conversion”
What is a type?

• Three points of view
  
  • Denotational: Set of values
  
  • Constructive: A type is “built-in” or “composite”
  
  • Abstraction-based: A type is an interface that defines a set of consistent operations
Denotation

• Under denotation, a value has a given type if it belongs to a set.

• An object has a type, if its value is guaranteed to be in a certain set.

• A set of values is called a domain (i.e., its type).

• Similar to enum in C
Built-in Types

• Built-in/primitive/elementary types
  • Mimic hardware units
    • e.g., boolean, character, integer, real (float)
  • Implementation varies across languages

• Characters are traditionally one-byte quantitates using the ASCII character set
Built-in Types: Unicode

- Newer languages have built-in characters that support Unicode character sets
- **Unicode is implemented using two-byte quantities.**
Built-in Types: Unicode

- Newer languages have built-in characters that support Unicode character sets
- **Unicode is implemented using two-byte quantities.**

This is very important for moving legacy code.
Built-in Types: Numeric Types

• Most languages support **integers and floats**
  • (Their value range is implementation dependent)

• Some languages support other numeric types
  • Complex Numbers (e.g., Fortran, Python)
  • Rational Number (e.g., scheme, common Lisp)
  • Signed and Unsigned integers (e.g., C, Modula-2)
  • Fixed point Numbers (e.g., Ada, Cobol)

• Some languages distinguish numeric types depending on their precision.
Composite

• A composite type is created by applying type constructors to simpler types
  • Records
  • Structs
  • Arrays
  • Sets
  • Classes
Classification of Types: Enumerations

- **Enumerations** improve program readability and error checking.

- First introduced in Pascal (but also exist in C):
  - type weekday = (sun, mon, tue, wed, thu, fri, sat);
  - They are **defined in order**, so they can be used in enumeration controlled loops.
Classification of Types: Subranges

- **Subranges** define a valid range of values for a variable.
  - e.g., Type test_score = 0..100;
- The improve **readability** and **error checking**
Classification of Types: Orthogonality

• Recall, orthogonality means that all features behaves consistently.
  • e.g., a=b always denotes assignment.

• This makes life much easier when reasoning about different types.
Now that we’ve discussed the basics of types, let’s go back to equivalence, compatibility and inference.
Type Checking

- **Type Equivalence**: When are the types of two values are the same?
- **Type Compatibility**: Can a value of A be used when type B is expected?
- **Type Inference**: What is the type of an expression, given the type of the operands?
Type Checking

- **Type Equivalence**: When are the types of two values are the same?
- **Type Compatibility**: Can a value of A be used when type B is expected?
- **Type Inference**: What is the type of an expression, given the type of the operands?
Type Equivalence

- Type Equivalence is defined in terms of structural and name equivalence.
Structural Equivalence

• Two types are structurally equivalent if they have the same components put together in the same way.

```c
typedef struct { int a, b; } foo1;
```

Equivalent!

```c
typedef struct {
    int a, b;
} foo2;
```
Structural Equivalence

- Two types are structurally equivalent if they have the same **components** put together **in the same way**

```
typedef struct{int a,b;} foo1
```

```
typedef struct{
  int b;
  int a;
}foo2
```

Equivalent? **No in ML, yes everywhere else.**
Structural Equivalence

typedef struct{
char *name;
char *addre;
int age;
} student;

typedef struct{
char *name;
char *addre;
int age;
} school;

... but probably not intentional.
Name Equivalence.

- **Name equivalence** assumes that two definitions with different names are not the same.
- Solves the “student-school” problem
Name Equivalence: Aliases

• Under name equivalence it is possible to define a new type via

```c
TYPE new_type = old_type;
```

• Such a construction is called an alias.
TYPE new_type = old_type;

• Two ways to interpret an alias:
  • **Strict name equivalence**
    • `New_type` is a different type than `old_type`.
  • **Loose name equivalence**
    • `New_type` is the same type as `old_type`.
Problem with Loose

```pascal
TYPE celsius_temp = REAL;
    farhen_temp = REAL;
VAR  c: celsius_temp;
    f: farhen_temp;
...
    f:=c;(* probably should be an error*)
```
Type Conversion

• A value of one type can be used in a context of another type using type conversion or type cast
Converting Type Cast

- Under a **converting type cast**, the underlying bits are changed

```c
int i;
float f = 3.4;
i = (int) f;
/* runtime */
```
Non-Converting Type Cast

Under a **Non-converting type cast**, the underlying bits are not altered.

```c
int i;
float f = 3.4;
i = *((int*) & f);
/* Compile time*/
```
Type Checking

• **Type Equivalence**: When are the types of two values are the same?

• **Type Compatibility**: Can a value of A be used when type B is expected?

• **Type Inference**: What is the type of an expression, given the type of the operands?
Type Compatibility

• Most languages do not require type equivalence in every context

• Two types T and S are compatible in Ada if any of the following conditions are true:
  • T and S are equivalent
  • T is a subtype of S
  • S is a subtype of T
  • T and S are arrays with the same number elements and same type of elements
Type Compatibility

- Type coercion allows a value of one type to be used in a context that expects another.

```c
short int s;
unsigned long int l;
...
s=l;
```
Type Compatibility

- Type coercion allows a value of one type to be used in a context that expects another.

```
short int s;
unsigned long int l;
...
s=l;
```

This makes the system type weaker.
Generic Reference Types

- It is often useful to have a **generic reference type** that can hold any type of object
  - in Java this is **Object**
  - In C and C++ this is **void *

```c
void* v;
int* i;
...
v=i;
```
Type Checking

- **Type Equivalence**: When are the types of two values the same?
- **Type Compatibility**: Can a value of A be used when type B is expected?
- **Type Inference**: What is the type of an expression, given the type of the operands?
Type Inference

• Usually the type of the *overall expression is easy*.  
• However, for **subranges** and **composite** objects is not so simple.
Subranges

type Atype = 0..20;
    Btype = 10..20;
var a: Atype;
    b: Btype;
...

What is the type of a+b?
Records

- **Records** (structs in C and C++) allow **for a collection of related data to be manipulated together**.

```c
struct foo{
    int a;
    int b;
}
```
Record: Memory Layout

- There may be **holes** in the allocation of memory.

```plaintext
type ore = record
   name : two_char;
   atom_num: integer;
   atom_weight: real;
   met: Boolean;
end;
```
Record: Memory Layout

- There may be holes in the allocation of memory.

```
type ore = record
   name : two_char;
   atom_num: integer;
   atom_weight: real;
   met: Boolean;
end;
```

Holes waste space and complicate comparisons.
Other arrangements

Packed

<table>
<thead>
<tr>
<th>name</th>
<th>atom_num</th>
<th>mber</th>
<th>atom_weight</th>
<th>met</th>
</tr>
</thead>
</table>

4 bytes

Rearranged

<table>
<thead>
<tr>
<th>name</th>
<th>met</th>
</tr>
</thead>
<tbody>
<tr>
<td>atom_num</td>
<td>atom_weight</td>
</tr>
</tbody>
</table>

4 bytes
Packed layouts require multiple instructions for accessing elements and assignments.
Variant Records

- A **variant record** (union) provides **two or more alternative fields** or collections of field **but only one bit** is valid at any given time.

```
struct element{
  char* Full_name;
  union{
    int atom_num;
    char atom_sym[2];
  }
}
```

Element can contain **atom_num** or **atom_sym**, but not both.
Variant Records

```c
struct element{
  char* Full_name;
  union{
    int atom_num;
    char atom_sym[2];
  }
}
```