A Type Is A Set

int n;

- Declaring a variable as a certain type restricts values to elements of a certain set
- A *type* is the set of values
  - plus a low-level representation
  - plus a collection of operations that can be applied to those values
A Tour Of Types

• There are too many to cover them all
• Instead, a short tour of the type menagerie
• Most ways you can construct a set in mathematics are also ways to construct a type in some programming language
• Tour organized around that connection
Outline

• Type Menagerie
  □ Primitive types
  □ Constructed types

• Uses For Types
  □ Type annotations and type inference
  □ Type checking
  □ Type equivalence issues
Primitive vs. Constructed Types

- **primitive type** - Any type that a program can use but cannot define for itself in the language
- **constructed type** - Any type that a programmer can define (using the primitive types)
- Some primitive types in ML: `int`, `real`, `char`
  - An ML program cannot define a type named `int` that works like the predefined `int`
- A constructed type: `int list`
  - Defined using the primitive type `int` and the `list` type constructor
Primitive Types

- The definition of primitive types part of language
- Some languages define the primitive types more strictly than others:
  - Some define the primitive types exactly (Java)
  - Others leave some wiggle room—the primitive types may be different sets in different implementations of the language (C, ML)
Comparing Integral Types

C:
char
unsigned char
short int
unsigned short int
int
unsigned int
long int
unsigned long int

Java:
byte (1-byte signed)
char (2-byte unsigned)
short (2-byte signed)
int (4-byte signed)
long (8-byte signed)

Scheme:
integer
Integers of unbounded range

No standard implementation, but longer sizes must provide at least as much range as shorter sizes.
Example – Size Matters

Haskell

fact 0 = 1
fact n = fact(n-1) * n;

> fact 100
933262154439441526816
992388562667004907159
682643816214685929638
952175999932299156089
414639761565182862536
979208272237582511852
10916864000000000000000
00000000000 :: Integer

C++

int fact(int n) {
    switch (n) {
    case 0 : return 1;
    default: return fact(n-1) * n;
    }
}

void main(void) {
    cout << fact(100);
}
0
Issues

• What sets do the primitive types signify?
  □ How much is part of the language specification, how much left up to the implementation?
  □ If necessary, how can a program find out? (\texttt{INT\_MAX} in C, \texttt{Int.maxInt} is \texttt{NONE} in ML or unlimited, etc.)

• What operations are supported?
  □ Detailed definitions: rounding, exceptions, etc.

• The choice of representation (e.g. bit size) is a critical part of these decisions
Outline

• Type Menagerie
  - Primitive types
  - Constructed types

• Uses For Types
  - Type annotations and type inference
  - Type checking
  - Type equivalence issues
Constructed Types

• Additional types defined based on language
• Enumerations, tuples, arrays, strings, lists, unions, subtypes, and function types
• For each one, there is connection between how *sets* are defined mathematically, and how *types* are defined in programming languages
Making Sets by Enumeration

- Mathematically, we can construct sets by just listing all the elements:

\[ S = \{a, b, c\} \]
Making Types by Enumeration

- Many languages support *enumerated types*:

  C: \[\text{enum coin} \{\text{penny, nickel, dime, quarter}\};\]
  Ada: \[\text{type GENDER is (MALE, FEMALE)};\]
  Pascal: \[\text{type primaryColors} = (\text{red, green, blue});\]
  Haskell: \[\text{data Day} = \text{M} | \text{Tu} | \text{W} | \text{Th} | \text{F} | \text{Sa} | \text{Su}\]

- These define a new type (= set)
- They also define a collection of named constants of that type (= elements)
Representing Enumeration Values

• A common representation is to treat the values of an enumeration as small integers
• This may even be exposed to the programmer, as it is in C:

```c
enum coin { penny = 1, nickel = 5, dime = 10, quarter = 25 };  
enum escapes { BELL = '\a', BACKSPACE = '\b', TAB = '\t',  
             NEWLINE = '\n', VTAB = '\v', RETURN = '\r' };  
```
Operations on Enumeration Values

• Equality test (Haskell):

```haskell
data Day = M | Tu | W | Th | F | Sa | Su
isWeekend x = (x == Sa || x == Su)
isWeekend W returns false
```

• If the integer nature of the representation is exposed, a language will allow some or all integer operations:

Pascal: for C := red to blue do P(C)

C: int x = penny + nickel + dime; is 16
Exercise 0

```haskell
data Day = M | Tu | W | Th | F | Sa | Su
isWeekend x = (x == Sa || x == Su)
isWeekend W returns false
```

1. Define Haskell enumeration `data` that correspond to the enumerated set:

   ```haskell
direction = { north, south, east, west}
```

2. Define C++ enumeration `datatype` that correspond to the enumerated set:

   ```cpp
direction = { north, south, east, west}
```

3. Define Haskell function `degree` that translates `north, south, east, west` to 90, 180, 0 and 270 respectively.
Defining Sets by Tupling

- The Cartesian product of two or more sets defines sets of tuples:

$$ S = X \times Y $$

$$ = \{(x, y) \mid x \in X \land y \in Y\} $$

- Example:

| A = {light, dark} | B = {red, blue, green} | A x B = { (light,red), (light,blue), (light, green), (dark,red), (dark,blue), (dark, green) } |

A = {light, dark}
B = {red, blue, green}
A x B = { (light,red), (light,blue), (light, green), (dark,red), (dark,blue), (dark, green) }
Exercise 0.1

What is the Cartesian product defined by $A \times B$ for the enumerated sets:

$A = \{\text{winter, summer}\}$
$B = \{\text{hot, cold}\}$;
Aggregate and Scalar types

• Aggregate – type composed of more than one value; tuples, records, arrays, sets, etc.
• Scalar – type composed of a single value; integers, reals, enumeration, etc.
Defining Types by Tupling

- Some languages support pure tuples (ML):

  ```ml
  fun get1 (x : real * real) = #1 x;
  ```

- Others, record types or tuples with named fields:

```c
struct complex {
  double rp;
  double ip;
};

float getip(complex x){
  return x.ip;
}

complex y = {1.0, 2.0};
double i = getip(y); i?
```

```ml
type complex = {
  rp : real,
  ip : real
};

fun getip (x:complex)= #ip x;

val y = {ip=1.0,rp=2.0};

val i = getip y; i?
```
Exercise 0.2

1. Define a ML record type `person` with `name`, `age`, `height` and `weight` attributes.

2. Assign person variable `President` values of “George”, 57, 70, 175.

3. Give a function `name` that returns the name value of a person.

4. Use function `name` to return the name of the `President`.

ML:

```ml
type complex = {
    rp : real,
    ip : real
};
fun getip (x:complex)= #ip x;
val y = {ip=1.0,rp=2.0};
val i = getip y;  
```
Representing Tuple Values

• A common representation is to just place the elements side-by-side in memory

• But there are lots of details:
  - in what order?
  - with “holes” to align elements (e.g. on word boundaries) in memory?
  - is any or all of this visible to the programmer?
Example: ANSI C

The members of a structure have addresses increasing in the order of their declarations. A non-field member of a structure is aligned at an addressing boundary depending on its type; therefore, there may be unnamed holes in a structure. If a pointer to a structure is cast to the type of a pointer to its first member, the result refers to the first member...

Adjacent field members of structures are packed into implementation-dependent storage units in an implementation-dependent direction...

*The C Programming Language*, 2nd ed.
Brian W. Kernighan and Dennis M. Ritchie
Operations on Tuple Values

• Construction:
  
  C: \[
  \text{complex } x = \{1.0, 2.0\};
  \]
  
  ML: \[
  \text{val } x = \{\text{ip}=1.0, \text{rp}=2.0\};
  \]

• Selection:
  
  C: \[
  x.\text{ip}
  \]
  
  ML: \[
  \#\text{ip } x
  \]

• Other operations depending on how much of the representation is exposed:
  
  C: \[
  \text{double } y = *((\text{double } \ast) \& x);
  \]
  
  struct person {
    char *firstname;
    char *lastname;
  }
  p1 = {"marcia","brady"};
Exercise 1

```
struct complex {
    double ip;
    double rp;
} x = {1.0, 2.0};

void main(void) {
    double *px = &x.ip;
    px++;
    cout << *px;
}
```

1. What is the value of `px` after: `double *px = &x.ip;`
2. After `px++`
3. What is the output?
Exercise 1 Continued

4. Define the C enum for Automobile of Ford, Honda, Yugo.
5. Define the Haskell data for Automobile of Ford, Honda, Yugo.
6. What are the Ford, Honda, Yugo values in C?
7. What are the Ford, Honda, Yugo values in Haskell?
8. What is the Cartesian product defined by $A \times B$:

\[
A = \{\text{Ford, Honda, Yugo}\}
\]
\[
B = \{\text{red, blue, green}\}
\]
Sets Of Vectors

- **Fixed-size vectors:** \( S = X^n \)

\[
= \left\{ (x_1, \ldots, x_n) \mid \forall i \cdot x_i \in X \right\}
\]

Example: \( X = \{3, 5\} \)

\( X^3 = X \times X \times X = \{(3,3,3), (3,3,5), (3,5,3), (3,5,5), (5,3,3), (5,3,5), (5,5,3), (5,5,5)\} \)

- **Arbitrary-size vectors:** \( S = X^* \)

\[
= \bigcup_{i} X^i
\]
Types Relate To Vectors

- Arrays, strings and lists
- Like tuples, but with many variations
- One example: indexes
  - What are the index values?
  - Is the array size fixed at compile time?
Index Values

- Java, C, C++:
  - First element of an array \( a \) is \( a[0] \)
  - Indexes are always integers starting from 0

- Pascal is more flexible:
  - Various index types are possible: integers, characters, enumerations, subranges
  - Starting index chosen by the programmer
  - Ending index too: size is fixed at compile time
Pascal Array Example

```pascal
type
    LetterCount = array['a'..'z'] of Integer;
    TempCount = array[-100..100] of Integer;
var
    Counts: LetterCount;

begin
    Counts['a'] = 1;
    TempCount[-12] = TempCount[-12] + 1;
    etc.
```
Types Related To Vectors

• Many variations on vector-related types:
  
  What are the index values?
  Is array size fixed at compile time (part of static type)?
  What operations are supported?
  Is redimensioning possible at runtime?
  Are multiple dimensions allowed?
  Is a higher-dimensional array the same as an array of arrays?
  What is the order of elements in memory?
  Is there a separate type for strings (not just array of characters)?
  Is there a separate type for lists?
struct complex {
    double ip;
    double rp;
};

void main(void) {
    int n;
    cin >> n;
    complex *x = new complex[n];
    for (int i=0; i<n; i++)
        x[i].ip = 1.0;
    cout << x[n-1].ip;
}
Exercise 2: Continued

a) The following is invalid C. Why?
b) How many variables are defined of type complex?
c) How many variables are defined of type double?
d) If \[ x[3][1][4] \] is followed by \[ x[3][1][5] \] in memory, what is the next index in memory? \[ x[\_][\_][\_] \]

```
struct complex {
    double ip;
    double rp;
};

void main(void) {
    complex x[5][3][6];
    x[3][1][4].rp = 2.0;
    x[3,2,4].ip = 1.0;
}
```
Making Sets by Union

- We can make a new set by taking the union of existing sets:

$$S = X \cup Y$$

- Example: \(A = \{a, b, c\}\)
  \(B = \{b, c, d\}\)
  \(A \cup B = \{a, b, c, d\}\)
Making Types by Union

• Many languages support **union** types:

  C:

  ```c
  union element {
    int i;
    float f;
  };
  ```

  ML:

  ```ml
  datatype element =
    I of int |
    F of real;
  ```

• One value accessible as different types.
Representing Union Values

- The two representations can overlap each other in memory

```c
union element {
    int i;
    char c;
} u;

/* sizeof(u) ==
   max(sizeof(u.i),sizeof(u.c)) */
```

- This representation may or may not be exposed to the programmer
- Example:
  ```c
  element x;
  x.i = 65;
  cout << x.c;    // 'A'
  ```
Strictly Typed Unions - ML

- Define union data type
- Construct
- Extract the contents by defining operations that return only one type of value in the union:

```ml
datatype element =
    I of int |
    F of real;

fun getReal (F x) = x        F to real
  | getReal (I x) = real x;    I to real

val y : element= F 3.0;
val z = getReal y;
y is F 3.0, z is 3.0

val a : element= I 4;
val b = getReal a;
a is I 4, b is 4.0
```
Example – Union

datatype element =
    I of int |
    F of real;

fun double (F x) = F(x*2.0)
    | double (I x) = I(x*2);

double (I 4);
    val it = I 8 : element

double (F 4.0);
    val it = R 8.0 : element

What is double (double (F 4.0))? 
What is double range? - domain?
Is strict type checking still in force?
Example – Union – Lists

Is \([I\ 5,\ F\ 5.0]\) valid?
Is \(\text{map}(\text{double},\ [F\ 4.3,\ I\ 5,\ F\ 3.0])\) valid?

\[
\begin{align*}
\text{datatype} & \quad \text{element} = \\
& \quad \text{I} \; \text{of} \; \text{int} \mid \\
& \quad \text{F} \; \text{of} \; \text{real}; \\
\text{fun} & \quad \text{double} \ (\text{F} \ x) = \text{F}(x*2.0) \\
& \quad \mid \text{double} \ (\text{I} \ x) = \text{I}(x*2); \\
\text{fun} & \quad \text{map} \ (f,\ []) = [] \\
& \quad \mid \text{map} \ (f,\ (h::t)) = (f\ h)::\text{map}(f,\ t); \\
\text{map} & \quad \text{(double,} \ [I\ 4,\ I\ 5,\ F\ 3.0]);
\end{align*}
\]

val it = \([I\ 8,\ I\ 10,\ F\ 6.0]\) : \text{element\ list}
Example – Union – Lists

datatype element =
  I of int |
  F of real;

fun add (I x, I y) = I(x + y)
|  add (F x, F y) = F(x + y)
|  add (F x, I y) = F(x + real y)
|  add (I x, F y) = F(real x + y);

fun sum [I x] = I x
|  sum [F x] = F x
|  sum ((I h)::t) = add(I h, sum t)
|  sum ((F h)::t) = add(F h, sum t);

sum [ F 5.3, I 6 ];
  val it = F 11.3 : element
Exercise 2.9 – Union – Lists

datatype element =
    I of int |
    F of real;

fun add (I x, I y) = I(x + y)
| add (F x, F y) = F(x + y)
| add (F x, I y) = F(x + real y)
| add (I x, F y) = F(real x + y);

fun sum [I x] = I x
| sum [F x] = F x
| sum ((I h)::t) = add(I h, sum t)
| sum ((F h)::t) = add(F h, sum t);

Can add be rewritten using three patterns?
Can sum be rewritten using three patterns?
Example – Union – Scalars & Lists

datatype element =
    F of real |
    Fl of real list;

val x = F 5.3;
    val x = F 5.3 : element

val x = Fl [5.3, 3.4, 4.3];
    val x = Fl [5.3, 3.4, 4.3] : element

fun add (F x, F y) = x + y
| add (F x, Fl []) = x
| add (F x, Fl (h::t)) = h + add (F x, Fl t);

add (F 3.0, Fl [5.3, 4.2]); returns 12.5
Loosely Typed Unions

- Some languages expose union implementation details
- Programs can take advantage of the fact that the specific type of a value is lost:

```c++
union element {
    int i;
    float f;
};

union element e;
e.i = 100;
int
float x = e.f;
float
cout << x;
```

On 32-bit Intel processor output is:

1.4013e-043
Exercise 3

union element {
    int i;
    char c;
};

void main(void) {
    union element e;
    e.c = 'A';
    int i = e.i;
    cout << i;
}

1. Is the output of the program:
   a) 65 — the integer value of ‘A’
   b) -858993599 — the integer value of ‘A’
   c) both a and b

2. What makes the difference?
Exercise 3

3. Write the function `add` of two `element` types and that returns the sum as an `element`.

4. What is the value of `x` at the end of execution?

```c
union element {
    int i;
    float f;
};

void main(void) {
    element x, y, z;
    y.i = 3;
    z.f = 4.0;
    x = add(y, z);
}
```
Exercise 3

datatype element =
    I of int |
    F of real;

fun getReal (F x) = x
    | getReal (I x) = real x; int to real

5. Write the `getInt` function for the `element` type.
6. Write a function to add two `element` types and return an `element`.
7. Use function `add` to add two `element` types.
What ANSI C Says About This

A union may be thought of as a structure all of whose members begin at offset 0 and whose size is sufficient to contain any of its members. At most one of the members can be stored in a union at any time. If a pointer to a union is cast to the type of a pointer to a member, the result refers to that member.

In general, a member of a union may not be inspected unless the value of the union as been assigned using that same member.

The C Programming Language, 2nd ed.
Brian W. Kernighan and Dennis M. Ritchie
A Middle Way: Variant Records

• Union where specific type is linked to the value of a field ("discriminated union")
• A variety of languages including Ada and Modula-2
Ada Variant Record Example

```ada
type DEVICE is (PRINTER, DISK);

type PERIPHERAL(Unit: DEVICE) is record
  HoursWorking: INTEGER;
  case Unit is
    when PRINTER =>
      Line_count: INTEGER;
    when DISK =>
      Cylinder: INTEGER;
      Track: INTEGER;
  end case;
end record;

DriveC : PERIPHERAL(DISK);
DriveC.Line_count = 5000;  Invalid - No Line_count
DriveC.Cylinder = 3;       Valid - Cylinder
```
Making Subsets

- We can define the subset selected by any predicate $P$:

$$S = \{ x \in X \mid P(x) \}$$

- $S$ is the set of elements $x$ from $X$ where $P(x)$ is true. $P$ is a filter function.
Exercise 3.1

\[ S = \{ x \in X \mid P(x) \} \]

\[ \text{predicate1} \ x = (x \mod 2) == 1 \]
\[ \text{predicate2} \ x = x < 3 \]

\[ \text{filter f} \ [\] = [] \]
\[ \text{filter f} \ (h:t) = \]
\[ \quad \text{if (f h) then h : (filter f t)} \]
\[ \quad \text{else (filter f t)} \]

• What is returned in the following?

\[ \text{filter predicate1} \ [1,2,3,4,5] \]
\[ \text{filter predicate2} \ [1,2,3,4,5] \]
Making Subtypes

- Some languages support subtypes, with more or less generality

- Less general: Pascal subranges
  ```pascal
type digit = 0..9;
```

- More general: Ada subtypes
  ```ada
type DAY is (MON, TUE, WED, THU, FRI, SAT, SUN);
subtype DIGIT is INTEGER range 0..9;
subtype WEEKDAY is DAY range MON..FRI;
```

- Most general: Lisp types with predicates
Example: Ada Subtypes

```ada
type DEVICE is (PRINTER, DISK);

type PERIPHERAL(Unit: DEVICE) is record
  HoursWorking: INTEGER;
  case Unit is
    when PRINTER =>
      Line_count: INTEGER;
    when DISK =>
      Cylinder: INTEGER;
      Track: INTEGER;
  end case;
end record;

subtype DISK_UNIT is PERIPHERAL(DISK);
```

A `DISK_UNIT` is a record of: `HoursWorking`, `Cylinder`, `Track`
Example: Lisp Types with Predicates

(declare (type integer x))
   x is an integer

(declare (type (and number (not integer)) x))
   x is a number and not an integer

(declare (type (and integer (satisfies evenp)) x))
   x is an integer and is even
Representing Subtype Values

• Usually, we just use the same representation for the subtype as for the supertype

• Questions:
  - Can the subtype occupy less memory? Does \( x: 1..9 \) take the same number of bits as \( x: \text{Integer} \)?
  - Do you enforce the subtyping? Is \( x := 10 \) legal? What about \( x := x + 1 \)?
Operations on Subtype Values

- Usually, supports all the same operations that are supported on the supertype
- And perhaps additional operations that would not make sense on the supertype:
  
  ```
  function toDigit(X: Digit): Char;
  
  ```

- Important meditation:

  A subtype is a subset of values, but support a superset of operations.

Example:

- Subtype: hours = 1 .. 12
- Operations: Predicates: AM, PM, Afternoon
  Special arithmetic: 9am+5=2pm
A Word About Classes

- Classes - key idea of object-oriented programming
- In class-based object-oriented languages, a class can be a type: data and operations on that data, bundled together
- A subclass is a specialization of a superset
- A subclass is a subtype: it includes a subset of the objects, but supports a superset of the operations
- More about this in Chapter 13
Making Sets of Functions

The set of functions $S$ that map values from domain $D$ into the range $R$:

$$S = D \rightarrow R$$

$$= \{ f \mid \text{dom } f = D \land \text{ran } f = R \}$$
Making Types of Functions

- Most languages have some notion of the type of a function:
- What is the **domain** and **range** of the following?

C: ```
int f(char a, char b) {
    return a==b;
}
```  

ML: ```
fun f(a:char, b:char) = (a = b);
```
Exercise 3.2

- For valid mappings, give the domain, range and signature of the following:

1. fun f1 (x : int) = x > ~1;
2. fun f2 (0 : int) = true
   | f2 (z : int) = z < ~1;
3. fun f3 (0 : int) = true
   | f3 (z : real) = z < ~1.0;
4. fun f4 [] = 0
   | f4 (h::t) = h + f4 t;
5. fun f5 [] = []
   | f5 (h::t) = (3*h) :: f5 t;
Operations on Function Values

- *Call* functions
- Take for granted that other types of values could be passed as parameters, bound to variables, and so on
- Not with function values: many languages support nothing beyond function call
- We will see more operations in ML
Outline

- Type Menagerie
  - Primitive types
  - Constructed types

- Uses For Types
  - Type annotations and type inference
  - Type checking
  - Type equivalence issues
Type Annotations – (e.g. float x)

- Many languages require (Java), or at least allow (ML), type annotations on variables, functions, ...
- The programmer uses them to supply static type information to the language system
- Also a form of documentation, and make programs easier for people to read
- Part of the language is syntax for describing types (think of *, -> and list in ML)
Intrinsic Types

• Some languages use naming conventions to declare the types of variables
  - Dialects of BASIC: \texttt{S$} is a string
  - Dialects of Fortran: \texttt{I} is an integer

• Like explicit annotations, these supply static type information to the language system and the human reader
Extreme Type Inference

- ML takes type inference to extremes
- Infers a static type for every expression and for every function
- Usually requires no programmer annotations
- What is the *domain* and *range* signature of the following?

```ml
fun f (x, y, z) = x + y * z;
```
Simple Type Inference

- Most languages require some simple kinds of type inference
- Constants usually have static types
  - Java: `10` has type `int`, `10L` has type `long`
- Expressions may have static types, inferred from operators and types of operands
  - Java: if `a` is `double`, `a*0` is `double (0.0)`
Outline

• Type Menagerie
  - Primitive types
  - Constructed types

• Uses For Types
  - Type annotations and type inference
  - Type checking
  - Type equivalence issues
Static Type Checking

- Static type checking determines a type for everything before execution (during compile) variables, functions, expressions, everything
- Compile-time error messages when static types are not consistent
  - Operators: \( 1 + "abc" \)
  - Functions: \( \text{round}("abc") \)
  - Statements: \( \text{if } "abc" \text{ then } ... \)
- Most modern languages are statically typed
Dynamic Typing

- In some languages, programs are not statically type-checked before being run
- Still *dynamically* type-checked usually
- At runtime, the language system checks that operands are of suitable types for operators

Example: \[ \text{function add}(x, y) = x + y; \]

\[
\begin{align*}
  z &= 3 + "4.2"; \quad \text{in VB is 7.2} \\
  z &= "3" + "4.2" \quad \text{in VB is 34.2}
\end{align*}
\]
Exercise 4 – Match the VB 6.0

a) fails with type mismatch
b) print Hello1
c) print 1
d) print 2

<table>
<thead>
<tr>
<th>Private Sub Picture1_Click()</th>
<th>Private Sub Picture1_Click()</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = x + 1</td>
<td>x = x + “Hello”</td>
</tr>
<tr>
<td>Picture1.Print x + &quot;1&quot;</td>
<td>Picture1.Print x + &quot;1&quot;</td>
</tr>
<tr>
<td>End Sub</td>
<td>End Sub</td>
</tr>
<tr>
<td>Private Sub Picture1_Click()</td>
<td>Private Sub Picture1_Click()</td>
</tr>
<tr>
<td>x = x + &quot;Hello&quot;</td>
<td>x = x + &quot;1&quot;</td>
</tr>
<tr>
<td>Picture1.Print x + 1</td>
<td>Picture1.Print x + 1</td>
</tr>
<tr>
<td>End Sub</td>
<td>End Sub</td>
</tr>
</tbody>
</table>
Exercise 4 – Match the C++

<table>
<thead>
<tr>
<th>a)</th>
<th>b)</th>
<th>c)</th>
<th>d)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compile error</td>
<td>print 2</td>
<td>print 1</td>
<td>print b</td>
</tr>
</tbody>
</table>

| char c = '1'; | char c = '1'; | char c = '1'; | char c = '1'; |
| c = c + '1'; | c = c + 1;  | c = c + 1.6; | c = c + 1.6; |
| cout << c;   | cout << c;  | cout << c;   | cout << c;   |
Example: Lisp

- This Lisp function adds two numbers:

  \[
  \text{(defun } f \ (a \ b) \ (+ \ a \ b))
  \]

- It won’t work if \texttt{a} or \texttt{b} is not a number

- An improper call, like \texttt{(f nil nil)}, is not caught at compile time

- It is caught at runtime – that is dynamic typing
Dynamic Typing Still Uses Types

- Although dynamic typing does not type everything at compile time, it still uses types.
- In a way, it uses them even more than static typing.
- It needs to have types to check at runtime.
- So the language system must store type information with values in memory.

Example: "abc" # "xyz"

Valid only if operation # defined on strings.
Static And Dynamic Typing

• Not quite a black-and-white picture
• Statically typed languages often use some dynamic typing
   Subtypes can cause this
   Everything is typed at compile time, but compile-time type may have subtypes
   At runtime, it may be necessary to check a value’s membership in a subtype
   This problem arises in object-oriented languages especially – more in Chapter 13
Static And Dynamic Typing

• Dynamically typed languages often use some static typing
  - Static types can be inferred for parts of Lisp programs, using constant types and declarations
  - Lisp compilers can use static type information to generate better code, eliminating runtime type checks
Explicit Runtime Type Tests

• Some languages allow explicit runtime type tests:
  - Java: test object class type with `instanceof` operator
  - Modula-3: branch on object type with `typecase` statement

• These require type information to be present at runtime, even when the language is mostly statically typed
Strong Typing, Weak Typing

• The purpose of type-checking is to prevent the application of operations to incorrect types of operands

• In some languages, like ML and Java, the type-checking is thorough enough to guarantee this—that’s strong typing

• Many languages (e.g. C and VB 6.0) fall short of this: there are holes in the type system that add flexibility but weaken the guarantee
Outline

• Type Menagerie
  □ Primitive types
  □ Constructed types

• Uses For Types
  □ Type declarations and inference
  □ Static and dynamic typing
  □ Type equivalence issues
Type Equivalence

- When are two types the same?
- An important question for static and dynamic type checking
- For instance, a language might permit assignment $a := b$ if $b$ has “the same” type as $a$
- Different languages decide type equivalence in different ways
Type Equivalence

- **Name equivalence**: types are the same if and only if they have the same name.

- **Structural equivalence**: types are the same if and only if they are built from the same primitive types using the same type constructors in the same order.

- Not the only two ways to decide equivalence, just the two easiest to explain.

- Languages often use odd variations or combinations.
Type Equivalence Example

```ml
type irpair1 = int * real; (int, real)
type irpair2 = int * real; (int, real)
fun f(x:irpair1) = #1 x;
```

- What happens if you try to pass `f` a parameter of type `irpair2`?
  - Name equivalence does not permit this: `irpair2` and `irpair1` are different names
  - Structural equivalence does permit this, since the types are constructed identically
- ML does permit it based on structure
Type Equivalence Example

```pascal
var
  Counts1: array['a'..'z'] of Integer;
  Counts2: array['a'..'z'] of Integer;
```

- What happens if you try to assign `Counts1 := Counts2`?
  - Name equivalence does not permit this: the types of `Counts1` and `Counts2` are unnamed
  - Structural equivalence does permit this, since the types are constructed identically
- Most Pascal systems do not permit it
Exercise 5

The following C code fails to compile. Does C use name or structural equivalence?

```c
struct complexA {
    double ip;
    double rp;
};

struct complexB {
    double ip;
    double rp;
};

void main(void) {
    complexA xA = { 1.0, 2.0 };;
    complexB xB;
    xB = xA;               \text{Error}
}
```
Conclusion

• A key question for type systems: how much of the representation is exposed?

• Some programmers prefer languages like C that expose many implementation details
  □ They offer the power to cut through type abstractions, when it is useful or efficient or fun to do so

• Others prefer languages like ML that hide all implementation details (abstract types)
  □ Clean, mathematical interfaces make it easier to write correct programs, and to prove them correct