

# Function Types

C, C++, and ML treat functions as just another data type that can be manipulated

- ☞ Functions can be passed as values; just as values that belong to other data types
- ☞ Functions belong to **function types**

Example: in ML consider the function type

$\text{real} \rightarrow \text{int}$

This type represents the set of all functions from real to int.

We have seen some members of this type:

floor:  $\text{real} \rightarrow \text{int}$

ceil:  $\text{real} \rightarrow \text{int}$

round:  $\text{real} \rightarrow \text{int}$

# Function Types

Example: Functions as values

```
- fun myfun (x:real):int = round(x);  
val myfun = fn:real -> int
```

```
- val foo = myfun;  
val foo = fn:real -> int
```

```
- foo(3.4);  
val it = 3 : int
```

Example: Functions as function arguments

```
- fun myfun(f:real -> int) = ...;  
- myfun(round);  
- myfun(ceil);
```

☞ A function is just an element of a particular function set.

## Why do we use types?

- Types allow the computer/language system to assist the developer write better programs. Type mismatches in a program usually indicate some sort of programming error.
  - Static type checking – check the types of all statements and expressions at compile time.
  - Dynamic type checking – check the types at runtime.

## Type Equivalence

- I. Name Equivalence – two objects are of the same type of and only if they share the same type name.

### Example: Java

```
Class Foobar {  
    int i;  
    float f;  
}
```

```
Class Goobar {  
    int i;  
    float f;  
}
```

```
Foobar o = new Goobar();
```

Error; even though the types look the same, their names are different, therefore, Java will raise an error.

☞ Java uses name equivalence

# Type Equivalence

II. Structural Equivalence – two objects are of the same type if and only if they share the same type structure.

Example: ML

```
- type person = int * int * string * string;  
- type mytuple = int * int * string * string;  
- val joe:person = (38, 185, "married", "pilot"):mytuple;
```

Even though the type names are different, ML correctly recognizes this statement.

☞ ML uses structural equivalence.

# Homework #4

HW #4

Problems: 6.2, 6.4 d through g,

**due Friday 2/24** in class.

# Patterns The Essence of Functional Programming

Up to now we have defined functions in a very traditional way:  
function name + variable name parameters

Read Chap 7

In functional programming we can exploit the structure of objects during a function definition/call by using patterns and pattern matching.

Example: no pattern matching, factorial  
- fun fact(x) = if x = 0 then 1 else x\*fact(x-1);

$$x! \begin{cases} 1 & \text{if } x = 0 \\ x*(x-1)! & \text{otherwise} \end{cases}$$

Example: with pattern matching, factorial  
- fun fact 0 = 1  
| fact n = n \* fact(n-1);

Very simple pattern: either it is 0 or not.

# Patterns

In order to use patterns we need to extend our ML syntax for function definitions:

```
<fun-def> ::= fun <fun-bodies>
<fun-bodies> ::= <fun-body>
| <fun-body> | <fun-bodies>
<fun-body> ::= <fun-name> <pattern> = <expression>
<pattern> ::= any function and operator free expression
(constructors are allowed).
```

## Valid Patterns:

1  
(a,b)  
[2,3]  
q::rest

## Invalid Patterns:

1+a  
f(q)