A Third Look At ML

Outline

- More pattern matching
- Function values and anonymous functions
- Higher-order functions and currying
- Predefined higher-order functions

More Pattern-Matching

- Last time we saw pattern-matching in function definitions:
  - ```
    fun f 0 = "zero"
    |   f _ = "non-zero";
    ```
- Pattern-matching occurs in several other kinds of ML expressions:
  - ```
    case n of
      0 => "zero" |
      _ => "non-zero";
    ```

Match Syntax

- A *rule* is a piece of ML syntax that looks like this:

  <*rule*> `::=` <*pattern*> **`=>`** <*expression*>

- A *match* consists of one or more rules separated by a vertical bar, like this:

  <*match*> `::=` <*rule*> `|` <*rule*> `'`**`|`**`'` <*match*>

- Each rule in a match must have the same type of expression on the right-hand side
- A match is not an expression by itself, but forms a part of several kinds of ML expressions

Case Expressions

```
- case 1+1 of
=     3 => "three" |
=     2 => "two" |
=     _ => "hmm";
val it = "two" : string
```

- The syntax is

  <*case-expr*> `::=` **`case`** <*expression*> **`of`** <*match*>
- This is a very powerful case construct—unlike many languages, it does more than just compare with constants

Example

```
case x of
  _::_::c::_ => c |
  _::b::_ => b |
  a::_ => a |
  nil => 0
```

The value of this expression is the third element of the list **x**, if it has at least three, or the second element if **x** has only two, or the first element if **x** has only one, or 0 if **x** is empty.

Generalizes **`if`**

```
if exp1 then exp2 else exp3
```

```
case exp1 of
  true => exp2 |
  false => exp3
```

- The two expressions above are equivalent
- So **`if`**-**`then`**-**`else`** is really just a special case of **`case`**

Outline

- More pattern matching
- Function values and anonymous functions
- Higher-order functions and currying
- Predefined higher-order functions

Predefined Functions

- When an ML language system starts, there are many predefined variables
- Some are bound to functions:

```
- ord;
val it = fn : char -> int
- ~;
val it = fn : int -> int
```

Defining Functions

- We have seen the **`fun`** notation for defining new named functions
- You can also define new names for old functions, using **`val`** just as for other kinds of values:

```
- val x = ~;
val x = fn : int -> int
- x 3;
val it = ~3 : int
```

Function Values

- Functions in ML *do not have names*
- Just like other kinds of values, function values may be given one or more names by binding them to variables
- The **`fun`** syntax does two separate things:
  - Creates a new function value
  - Binds that function value to a name

Anonymous Functions

- Named function:

```
- fun f x = x + 2;
val f = fn : int -> int
- f 1;
val it = 3 : int
```

- Anonymous function:

```
- fn x => x + 2;
val it = fn : int -> int
- (fn x => x + 2) 1;
val it = 3 : int
```

The `fn` Syntax

- Another use of the match syntax

  <*fun-expr*> `::=` **`fn`** <*match*>
- Using **`fn`**, we get an expression whose value is an (anonymous) function
- We can define what **`fun`** does in terms of **`val`** and **`fn`**
- These two definitions have the same effect:
  - **`fun f x = x + 2`**
  - **`val f = fn x => x + 2`**

Using Anonymous Functions

- One simple application: when you need a small function in just one place
- Without **fn**:

```
- fun intBefore (a,b) = a < b;
val intBefore = fn : int * int -> bool
- quicksort ([1,4,3,2,5], intBefore);
val it = [1,2,3,4,5] : int list
```

- With **fn**:

```
- quicksort ([1,4,3,2,5], fn (a,b) => a<b);
val it = [1,2,3,4,5] : int list
- quicksort ([1,4,3,2,5], fn (a,b) => a>b);
val it = [5,4,3,2,1] : int list
```

The **op** keyword

```
- op *;
val it = fn : int * int -> int
- quicksort ([1,4,3,2,5], op <);
val it = [1,2,3,4,5] : int list
```

- Binary operators are special functions
- Sometimes you want to treat them like plain functions: to pass **`<`**, for example, as an argument of type **`int * int -> bool`**
- The keyword **`op`** before an operator gives you the underlying function

Outline

- More pattern matching
- Function values and anonymous functions
- Higher-order functions and currying
- Predefined higher-order functions

Higher-order Functions

- Every function has an *order*:
  - A function that does not take any functions as parameters, and does not return a function value, has *order 1*
  - A function that takes a function as a parameter or returns a function value has *order n+1*, where *n* is the order of its highest-order parameter or returned value
- The **`quicksort`** we just saw is a second-order function

Practice

What is the order of functions with each of the following ML types?

```
int * int -> bool
int list * (int * int -> bool) -> int list
int -> int -> int
(int -> int) * (int -> int) -> (int -> int)
int -> bool -> real -> string
```

What can you say about the order of a function with this type?

```
('a -> 'b) * ('c -> 'a) -> 'c -> 'b
```

Currying

- We've seen how to get two parameters into a function by passing a 2-tuple:

```
fun f (a,b) = a + b;
```

- Another way is to write a function that takes the first argument, and returns another function that takes the second argument:

```
fun g a = fn b => a+b;
```

- The general name for this is *currying*

Curried Addition

```
- fun f (a,b) = a+b;
val f = fn : int * int -> int
- fun g a = fn b => a+b;
val g = fn : int -> int -> int
- f(2,3);
val it = 5 : int
- g 2 3;
val it = 5 : int
```

- Remember that function application is left-associative
- So **`g 2 3`** means **`((g 2) 3)`**

Advantages

- No tuples: we get to write **`g 2 3`** instead of **`f(2,3)`**
- But the real advantage: we get to specialize functions for particular initial parameters

```
- val add2 = g 2;
val add2 = fn : int -> int
- add2 3;
val it = 5 : int
- add2 10;
val it = 12 : int
```

Advantages: Example

- Like the previous **`quicksort`**
- But now, the comparison function is a first, curried parameter

```
- quicksort (op <) [1,4,3,2,5];
val it = [1,2,3,4,5] : int list
- val sortBackward = quicksort (op >);
val sortBackward = fn : int list -> int list
- sortBackward [1,4,3,2,5];
val it = [5,4,3,2,1] : int list
```

Multiple Curried Parameters

- Currying generalizes to any number of parameters

```
- fun f (a,b,c) = a+b+c;
val f = fn : int * int * int -> int
- fun g a = fn b => fn c => a+b+c;
val g = fn : int -> int -> int -> int
- f (1,2,3);
val it = 6 : int
- g 1 2 3;
val it = 6 : int
```

Notation For Currying

- There is a much simpler notation for currying (on the next slide)
- The long notation we have used so far makes the little intermediate anonymous functions explicit

```
fun g a = fn b => fn c => a+b+c;
```

- But as long as you understand how it works, the simpler notation is much easier to read and write

Easier Notation for Currying

- Instead of writing:

```
fun f a = fn b => a+b;
```

- We can just write:

```
fun f a b = a+b;
```

- This generalizes for any number of curried arguments

```
- fun f a b c d = a+b+c+d;
val f = fn : int -> int -> int -> int -> int
```

Outline

- More pattern matching
- Function values and anonymous functions
- Higher-order functions and currying
- Predefined higher-order functions

Predefined Higher-Order Functions

- We will use three important predefined higher-order functions:
  - **`map`**
  - **`foldr`**
  - **`foldl`**
- Actually, **`foldr`** and **`foldl`** are very similar, as you might guess from the names

The **`map`** Function

- Used to apply a function to every element of a list, and collect a list of results

```
- map ~ [1,2,3,4];
val it = [~1,~2,~3,~4] : int list
- map (fn x => x+1) [1,2,3,4];
val it = [2,3,4,5] : int list
- map (fn x => x mod 2 = 0) [1,2,3,4];
val it = [false,true,false,true] : bool list
- map (op +) [(1,2),(3,4),(5,6)];
val it = [3,7,11] : int list
```

The **map** Function Is Curried

```
- map;
val it = fn : ('a -> 'b) -> 'a list -> 'b list
- val f = map (op +);
val f = fn : (int * int) list -> int list
- f [(1,2),(3,4)];
val it = [3,7] : int list
```

The `foldr` Function

- Used to combine all the elements of a list
- For example, to add up all the elements of a list **`x`**, we could write **`foldr (op +) 0 x`**
- It takes a function *f*, a starting value *c*, and a list *x* = [x_1, …, x_n] and computes:

$$f\left(x_1, f\left(x_2, \cdots f\left(x_{n-1}, f\left(x_n, c\right)\right)\cdots\right)\right)$$

- So **`foldr (op +) 0 [1,2,3,4]`** evaluates as 1+(2+(3+(4+0)))=10

Examples

```
- foldr (op +) 0 [1,2,3,4];
val it = 10 : int
- foldr (op * ) 1 [1,2,3,4];
val it = 24 : int
- foldr (op ^) "" ["abc","def","ghi"];
val it = "abcdefghi" : string
- foldr (op ::) [5] [1,2,3,4];
val it = [1,2,3,4,5] : int list
```

The **foldr** Function Is Curried

```
- foldr;
val it = fn : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
- foldr (op +);
val it = fn : int -> int list -> int
- foldr (op +) 0;
val it = fn : int list -> int
- val addup = foldr (op +) 0;
val addup = fn : int list -> int
- addup [1,2,3,4,5];
val it = 15 : int
```

The **`foldl`** Function

- Used to combine all the elements of a list
- Same results as **`foldr`** in some cases

```
- foldl (op +) 0 [1,2,3,4];
val it = 10 : int
- foldl (op * ) 1 [1,2,3,4];
val it = 24 : int
```

The `foldl` Function

- To add up all the elements of a list **`x`**, we could write **`foldl (op +) 0 x`**
- It takes a function *f*, a starting value *c*, and a list $x = [x_1, \ldots, x_n]$ and computes:

$$f\left(x_n, f\left(x_{n-1}, \cdots f\left(x_2, f\left(x_1, c\right)\right)\cdots\right)\right)$$

- So **`foldl (op +) 0 [1,2,3,4]`** evaluates as 4+(3+(2+(1+0)))=10
- Remember, **`foldr`** did 1+(2+(3+(4+0)))=10

The **`foldl`** Function

- **`foldl`** starts at the **`l`**eft, **`foldr`** starts at the **`r`**ight
- Difference does not matter when the function is associative and commutative, like **`+`** and **`*`**
- For other operations, it does matter

```
- foldr (op ^) "" ["abc","def","ghi"];
val it = "abcdefghi" : string
- foldl (op ^) "" ["abc","def","ghi"];
val it = "ghidefabc" : string
- foldr (op -) 0 [1,2,3,4];
val it = ~2 : int
- foldl (op -) 0 [1,2,3,4];
val it = 2 : int
```