>>

Higher-order Functions

∧ >>

❖ Higher-order Functions

Recall the quicksort library function

```
void qsort(void *base, size_t nelems, size_t width,
           int (*compare)(const void *, const void *));
```

Sorts an array of items of any type (hence **`(void *)`**)

- **`base`** is the address of the first element
- **`nelems`** is the number of elements in the array
- **`width`** is the size (in bytes) of each element
- **`compare`** is a function to compare two items
  - parameters are *pointers to items* to be compared
  - **`compare(`**a, b**`)`** returns -ve if a < b, +ve if a > b, 0 if equal

Example call: **`qsort(myArray, 10, sizeof(char *), strcmp)`**

<< ^ >>

❖ ... Higher-order Functions

`qsort()` is an example of a higher-order function

- a function that has parameters which are also functions

Functional programming languages support higher-order functions

- intrinsic to the language (functions are first-class citizens)
- used extensively to express algorithms concisely
- e.g. Haskell, Curry, Miranda, Lisp, Scheme, ML, Ruby, even Javascript

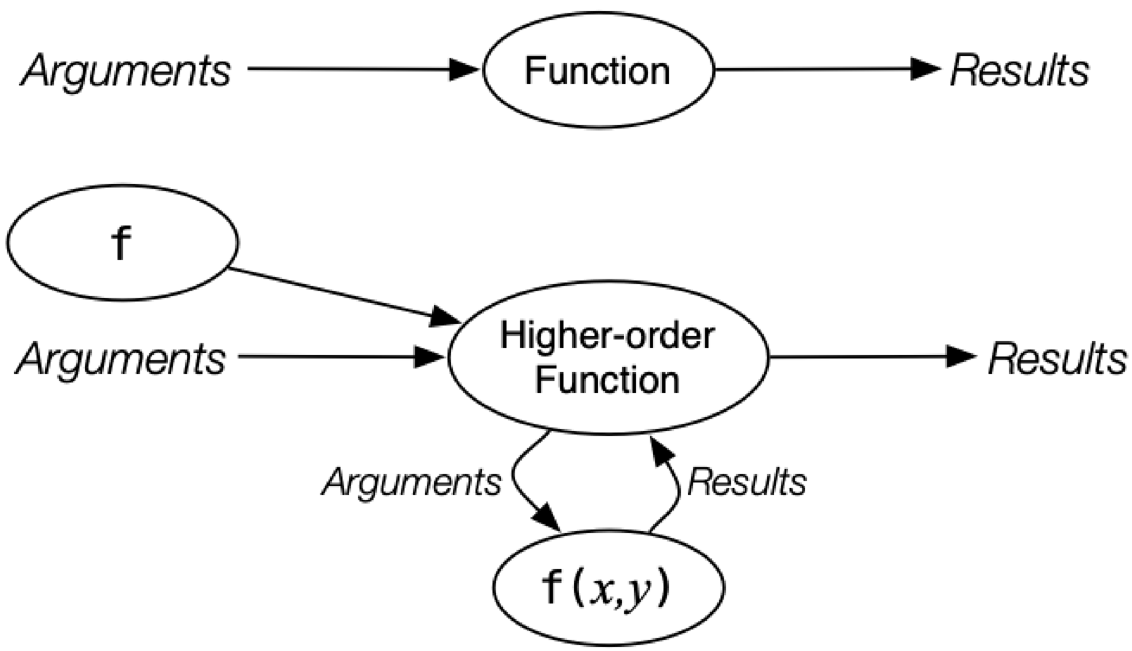
A useful place to explore higher-order functions: (recursive) functions on lists

Note that recursive functions are not necessarily higher-order, but they can be

<< ʌ >>

❖ ... Higher-order Functions

Regular functions vs Higher-order functions (in C)



COMP2521 20T2 ◊ Higher-order Functions [3/21]

<< ∧ >>

❖ Pointers to Functions

C is definitely not a functional language

- but somewhat supports higher-order functions via function pointers

Consider **`int (*compare)(const void *, const void *)`**

- **`compare`** is a pointer to a function
- that takes two **`(void *)`** pointers
- and returns an **`int`** result

The **`const`** declaration doesn't change the meaning

- but signals that the function should treat the arguments as read-only

<< ^ >>

❖ ... Pointers to Functions

How to get a value which is a function pointer ...

- write just the function name e.g. **`qsort`**, **`strcmp`**
- write the function name preceded by **`&`**, e.g. **`&qsort`**

Writing a function name followed by a parenthesis '**`(`**'

- is a *call* to the named function
- passing the supplied values as parameters

Example: **`strcmp("abc","def")`**, **`fgets(buf,10,stdin)`**, ...

- ... and pretty much every other function you've called in C

<< ∧ >>

❖ ... Pointers to Functions

Function pointers ...

- are references to memory address of a function
- are pointer values and can be assigned/passed

Example of use:

```
//define a function pointer variable "fp"
int (*fp)(List);
// assign a value to this variable
fp = &length;
// apply the function being pointed to
n = (*fp)(L);    // same as n = length(L)
```

<< ∧ >>

❖ Recursive functions on Lists

For this discussion, we define **`List`** as:

```
// a list node contains an integer value
typedef struct _node {
   int val;
   struct _node *next;
} Node;
// a List is a pointer to its first Node
typedef Node *List;
// we also uses Sedgewick's definition
typedef Node *Link;
```

The difference between **`List`** and **`Link`**

- type-wise they are identical
- **`List`** used for pointer to first node in a list
- **`Link`** used for pointer to some node in a list

<< ^ >>

❖ ... Recursive functions on Lists

The following functions help with list manipulation:

```
// returns a new empty List
List new();
// checks whether L is empty
bool empty(List L);
// returns first element in L
int head(List L);
// returns all but first element in L
List tail(List L);
// returns new list with x as head and L as tail
List insert(int x, List L);
// returns new list with x as last element
List append(List L, int x);
// returns new list which is concatenation of lists L1 and L2
List concat(List L1, List L2);
```

❖ Folding Lists

Example: determining the length of a list

```
// empty list has length 0
// a non-empty list has at least one element
//  plus as many as are in the rest of the list

int length(List L)
{
   if (empty(L))
     return 0;  // base case
   else
     return 1 + length(tail(L));
}
```

Exercise: write an iterative version using the empty/head/tail operations

<< ^ >>

❖ ... Folding Lists

Example: sum of values in a list

```
// empty list has sum 0
// a non-empty list has at least one element
//  plus the sum of the rest of the list

int sum(List L)
{
   if (empty(L))
     return 0;   // base case
   else
     return head(L) + sum(tail(L));
}
```

Note similar structure to **`length()`**

<< ^ >>

❖ ... Folding Lists

`sum()` and **`length()`** are examples of ...

- a common pattern of computation
- that reduces a list to a single value

This is commonly called **`fold()`** and defined as

```
int fold(List L, int (*f)(int x, int y), int id)
```

- **`L`** is the list to be reduced
- **`f`** is a function that takes two **`int`**s and returns an **`int`**
- **`id`** is the identity for the **`f`** function

❖ ... Folding Lists

The **sum** function could be defined using **fold**

```
int add(int x, int y) { return x+y; }
int sum(List L) { return fold(L, add, 0); }
```

We could define **product** using the same approach

```
int mult(int x, int y) { return x*y; }
int product(List L) { return fold(L, mult, 1); }
```

<< ∧ >>

❖ ... Folding Lists

The **`fold`** function is defined as

```
// compute f(L1, f(L2, f(L3, ... f(Ln,id))))

int fold(List L, int (*f)(int x, int y), int id)
{
   if (empty(L))
      return id;
   else
      return f( head(L), fold(tail(L),f,id) );
}
```

Example: **`fold([1,2,3],add,0)`** computes
`add(1,add(2,add(3,0)))`

❖ Mapping a List

Example: print items in a list (one per line)

```
// print first item, on a line by itself
// then print the rest of the items

void putList(List L) {
   if (!empty(L)) {
     showItem(head(L));
     putList(tail(L));
   }
}
```

<< ∧ >>

❖ ... Mapping a List

Example: doubling each item in a list

```
// apply a function to each item in a list

void doubleUp(List L)
{
   if (!empty(L)) {
      head(L) = 2 * head(L);
      doubleUp(tail(L));
   }
}
```

Notes:

- **`#define head(L) L->val`**
- modifies each **`Node`** in the list

<< ^ >>

❖ ... Mapping a List

`printList()` and **`doubleUp()`** are examples of ...

- a common pattern of computation
- that applies a function to each node in a list
- one difference: only **`doubleUp`** changes the node

This is commonly called **`map()`** and defined as

```
void map(List L, void (*f)(Link x));
```

- **`L`** is the list to be mapped
- **`f`** is a function that operates on a node

<< ^ >>

❖ ... Mapping a List

Defining **doubleUp** using **map**

```
void timesTwo(Link n) { n->val = n->val * 2; }
void doubleUp(List L) { map(L, timesTwo); }
```

Defining **printList** using **map**

```
void showItem(Link n) { printf("%d\n",n->val); }
void printList(List L) { map(L, showItem); }
```

<< ∧ >>

❖ ... Mapping a List

The **map** function could be defined as

```
void map(List L, void (*f)(Link x))
{
   if (!empty(L)) {
      f(L);
      map(tail(L), f);
   }
}
```

❖ ... Mapping a List

A variation on **`map`** returns a new list

```
List mapp(List L, int (*f)(int x))
{
   if (empty(L))
      return new();
   else
      return insert( f(head(L)), mapp(tail(L),f) );
}
```

Reminder:
`insert(int x, List L)` puts a new node containing **`x`** at the front of the list

<< ^ >>

❖ Example: list of fibs

If we had the following two functions:

```
// returns a list [1,2,3,...,n]
List seq(int n) { ... }
// returns n'th fibonacci number
int fibonacci(int n) { ... }
```

Then we could build a list of the first n Fibonacci numbers as

```
List firstNfib(int n) { return map(seq(n), fibonacci); }
```

❖ Example: length of a List

A function that computes the length of a list by

- replacing each value in the list by 1
- summing the elements of the new list

```
int one(int x) { return 1; }
int add(int x, int y) { return x+y; }

int length(List L)
    { return fold(mapp(L,one), add, 0); }
```

Produced: 7 Aug 2020