Распространённые функции высшего порядка

В предыдущем уроке мы начали разговор о функциях высшего порядка. Сейчас мы изучим несколько распространённых функций высшего порядка и поймём, откуда они появились.

Для выполнения этого задания на локальной машине вы можете использовать заготовку.

Функция `foreach` запускает функцию-аргумент на всех элементах списка по очереди, в цикле. Реализуйте её, а затем с её помощью реализуйте `list_print`. Напоминаем, что эта функция выводит список, и после каждого элемента добавляет пробел.

1task – program

Функция `map_mut` запускает функцию-аргумент на всех элементах списка и перезаписывает их. Реализуйте её, а затем с её помощью реализуйте `list_triple`, умножающий каждый элемент списка на 3.

2task – program

Функция `map` запускает функцию-аргумент на всех элементах списка и таким образом формирует новый список. Старый список остаётся нетронутым. Реализуйте `map`, а затем с её помощью реализуйте `list_copy`, копирующий список как есть, и `list_abs`, берущий модуль всех чисел в списке.

3task – program

Свертки

Рассмотрим функции поиска минимума и максимума в непустом массиве:

Минимум	Максимум

`int64_t array_min( int64_t* array, size_t sz ) {`
 	
`  int64_t result = INT64_MAX;`
	`int64_t array_max( int64_t* array, size_t sz ) {`
	
	`  int64_t result = INT64_MIN;`

```
    for( size_t i = 0; i < sz; i = i + 1 ) {

        result = min( result, array[i] );

    }


    return result;

}
```

```
    for( size_t i = 0; i < sz; i = i + 1 ) {

        result = max( result, array[i] );

    }


    return result;

}
```

Здесь `INT64_MAX` и `INT64_MIN` это максимальное и минимальное знаковое число в 64 разрядах.

Обе эти функции имеют "рабочее" значение `result`, называемое также *аккумулятор*. В начале выполнения он инициализируется фиксированным значением, затем мы проходимся по массиву и для каждого элемента единообразно пересчитываем значение аккумулятора.

Эти функции действуют по одинаковому шаблону и различаются только начальным значением переменной `result` и тем, как аккумулятор пересчитывается на каждой итерации.

Рассмотрим ещё несколько примеров:

Сумма	Слияние строчек

`int64_t array_sum( int64_t* array, size_t sz ) {`
 	
`    int64_t result = 0;`
	`// Конкатенация строк; выделяем память под новую строку`
	
	`// и записываем туда строки x и y через пробел;`
	
	`// деаллоцируем x`
	
	`char* string_concat(char* x, const char* y);`
	
	`char* array_to_string( char** array, size_t sz ) {`

```
    for( size_t i = 0; i < sz; i =
i + 1 ) {

        result = result + array[i] ;

    }


    return result;

}
```

```
    char* result = NULL;


    for( size_t i = 0; i < sz; i = i + 1 )
{

        result = string_concat( result,
array[i] );

    }


    return result;

}


//Результат: для массива {1,2,3}

//строчка  "1 2 3 "
```

Удивительно, но поднявшись на уровень абстракции выше, мы нашли другие функции, которые подходят под этот шаблон! В примере со слиянием строчек аккумулятор – строка, а элемент массива – число, так что не обязательно тип аккумулятора совпадает с типом элементов массива.

Свертки

Функции, которые проходят по большой структуре данных, накапливая результат, называются *свёртками (fold)*.

Напишем свертку для массивов:

```
struct array_int { int64_t* data; size_t size; };
```

```
/* Для читаемости определим новый тип функции для свертки */

typedef int64_t folding(int64_t, int64_t);
```

```
int64_t array_fold( const struct array_int array, int64_t init, folding f) {

  int64_t result = init;

  for( size_t i = 0; i < array.size; i = i + 1 ) {
    result = f( result, array.data[i] );
  }

  return result;
}
```

Выразим с её помощью сумму и поиск минимального элемента:

```
/* Минимальный элемент массива */
int64_t min( int64_t x, int64_t y) { if (x < y) {return x;} else {return y;} }

int64_t array_min( const struct array_int array ) {
  return array_fold( array, INT64_MAX, min);
}

/* Сумма элементов массива */
int64_t add( int64_t x, int64_t y) { return x + y; }

int64_t array_sum( const struct array_int array ) {
  return array_fold( array, 0, add );
}
```

```
/* Пример использования */

int main() {

  int64_t array_data[] = {1, 2, 3, -9, 5};

  const struct array_int array = { array_data,
sizeof(array_data)/sizeof(array_data[0]) };



  print_int64( array_min( array ) ) ;

  print_newline();

  print_int64( array_sum( array ) ) ;

  return 0;


}
```

Теперь реализуем функцию `fold`, которая сворачивает список так же, как мы делали с массивом.

4task – program

Наконец, напишем функцию `iterate`, которая сгенерирует список применяя функцию к первому элементу 0, 1, 2 раза и т.д. пока список не достигнет требуемой длины.

Например, возьмём функцию $f(x) = 3x$. Положим начальный элемент равным 2. Тогда список из пяти элементов будет выглядеть так: 2, 6, 18, 54, 162.

5task – program