Ветвящаяся рекурсия

А. Г. Фенстер, `fenster@fenster.name`

19 марта 2010 г.

Конспект семинара №6 по программированию для студентов 1 курса ММФ НГУ. Помните, что чтение конспекта не делает посещение соответствующего семинара необязательным! О найденных ошибках и неточностях, пожалуйста, пишите мне по адресу `fenster@fenster.name`. Также буду рад получить любые комментарии по поводу этого текста.

> Чтобы понять рекурсию, нужно понять рекурсию.
> *Автор неизвестен*

Прежде чем читать этот конспект, рекомендую вспомнить, как устроена функция «разворота ввода» `reverse` (см. конспект второго семинара).

1 Виды рекурсии

При выходе из функции управление передаётся в то место в коде, откуда функция была вызвана. В случае «простых» функций это не вызывает вопросов: например, обычно нет сомнений, что после успешного завершения функции `printf` программа будет выполняться дальше. То же самое верно и для рекурсивных функций.

Мы уже говорили, что среди всех рекурсивных функций (функций, вызывающих тем или иным способом себя) можно выделить функции, построенные по схеме *хвостовой рекурсии*: это такие функции, которые вызывают себя только перед самым выходом из функции (в инструкции `return`). Рекурсивная функция вычисления факториала, например,

представляет собой пример хвостовой рекурсии[1]; её последняя инструкция выглядит так:

```
return n * fact(n - 1);
```

Известно, что хвостовую рекурсию можно реализовать итеративно (циклом), используя лишь фиксированное количество дополнительных переменных (не зависящее от глубины рекурсии). И факториал, и бинарный поиск (тоже хвостовая рекурсия, хоть и чуть более сложная) можно и нужно реализовывать именно при помощи цикла, потому что рекурсивные (да и любые другие) вызовы функций увеличивают как время работы программы, так и потребляемую ей память.

Для понимания хвостовой рекурсии, в общем-то, не обязательно чётко понимать, что после вычисления рекурсивного вызова управление вернётся обратно в экземпляр функции, из которой вызов был сделан: всё равно вызывающая функция уже практически завершилась.

С другими видами рекурсии всё несколько сложнее. Добавление лишь одной инструкции после рекурсивного вызова (как в функции `reverse`) может привести к тому, что функция уже не будет преобразовываться в цикл без использования дополнительной памяти, и функция `reverse` — хороший пример этого. Но в функции `reverse` по крайней мере каждый экземпляр функции делал не более одного рекурсивного вызова.

Ветвящаяся рекурсия возникает, когда функция может вызвать себя несколько раз. На этом семинаре мы рассмотрим несколько примеров задач, в которых возникает такая рекурсия, поскольку в будущем (например, при изучении быстрой сортировки и алгоритмов на графах) такие функции будут встречаться нам часто.

[1]На самом деле это не совсем верно. Факториал будет являться действительно хвостовой рекурсией, если модифицировать его так:

```
int fact(int n, int accumulator)
{
    if (n == 0) return accumulator;
    return fact(n - 1, n * accumulator);
}
```

Корректно можно определить хвостовой вызов так: это вызов функции `g` из функции `f`, при котором результат функции `g` немедленно возвращается в качестве результата функции `f`. Если `f` и `g` — одна и та же функция, имеем хвостовую рекурсию (как в приведённом факториале с аккумулятором). Для простоты я намеренно использую не совсем верное определение хвостовой рекурсии.

2 Печать всех подмножеств

Задача. Напечатайте все подмножества множества $\{1, \ldots, N\}$.

При разработке рекурсивной функции необходимо придумать способ разбиения задачи на подзадачи меньшего размера. Самая «мелкая» подзадача должна решаться элементарно — это будет *терминальной ветвью* рекурсии (для факториала это вариант `if (n == 0) return 1;`).

Очевидно, что искомых подмножеств ровно 2^N штук: первый элемент может включаться или не включаться в подмножество (два варианта), второй элемент может включаться или не включаться (ещё два) и так далее для каждого из N элементов исходного множества.

Это соображение даёт способ подойти к разбиению задачи на подзадачи. Множество всех подмножеств множества $\{1, \ldots, N\}$ образовано объединением двух множеств: множества всех подмножеств $\{1, \ldots, N\}$, включающих элемент 1, и всех подмножеств, не включающих его. Если мы научимся находить все подмножества множества $\{2, \ldots, N\}$ (их 2^{N-1} штук), то из них мы можем получить искомые 2^N подмножеств, добавив и не добавив единицу к каждому из 2^{N-1}.

Как реализовать это на языке программирования? Будем представлять множество в виде массива из N элементов, в котором элементам, присутствующим во множестве, соответствует единица, а не присутствующим — ноль. Массивы у нас нумеруются от нуля, так что реально нам нужно либо перенумеровать элементы, либо использовать массив размера $N + 1$:

```
int M[N + 1];
```

Пусть функция `f` будет принимать решение по поводу элемента k. Описанная выше логика переносится на язык программирования почти без изменений:

```
M[k] = 0; /* не берём элемент k */
f(k + 1);
/* точка возврата из рекурсии */

M[k] = 1; /* а теперь берём элемент k */
f(k + 1);
```

Важно: вспомните то, о чём говорилось в самом начале. При возврате из функции `f(k + 1)` будет продолжено выполнение функции `f`.

Образуется дерево вызовов: экземпляр функции `f` вызывает `f` дважды, каждый из «потомков» делает то же самое, и так далее.

Осталось придумать терминальную ветвь. Очевидно, ничего не нужно делать, если принято решение по поводу каждого из N элементов:

```
if (k > N)
{
    print(M, N); /* каким-то образом печатаем множество */
    return;      /* выход из функции */
}
```

Напишем функцию `print` и полный вариант функции `f`. Глобальные переменные — зло, поэтому будем передавать все данные через параметры.

```
void print(int *M, int N)
{
    int i;
    printf("{ ");
    for (i = 1; i <= N; i++)
    {
        if (M[i])
        {
            printf("%d ", i);
        }
    }
    printf("}\n");
}

void f(int *M, int N, int k)
{
    if (k > N)
    {
        print(M, N);
        return;
    }
    M[k] = 0;
    f(M, N, k + 1);
    M[k] = 1;
    f(M, N, k + 1);
}
```

3 Разложение числа на множители

Задача. Напечатайте все варианты разложения данного натурального числа N на множители.

В этой задаче функция «разложить число k» должна, во-первых, выдать вариант «k» (оставить число не разложенным), а во-вторых, перебрать все делители числа k, большие единицы и меньшие k, и вызвать рекурсивно их разложение на множители.

Функции необходимо будет передавать параметром массив уже найденных до данного шага делителей.

4 Генерация перестановок

Задача. Найдите все перестановки элементов множества $\{1, \ldots, N\}$.

В этой задаче рекурсивная функция будет принимать два массива: начало построенной перестановки и множество неиспользованных элементов. На каждом шаге функция должна взять поочерёдно каждый из неиспользованных элементов, добавить его к построенной перестановке и вызвать себя для полученного варианта. Начальный вызов — перестановка пуста, все элементы не использованы; терминальная ветвь — все элементы использованы (в этом случае необходимо напечатать построенную перестановку).

Я постараюсь найти время и описать эту и предыдущую задачу более подробно.

5 Проверочное задание («пятиминутка»)

Тема: определить, что печатает данная рекурсивная функция (см. задание).