ДЕКЛАРАТИВНОЕ ПРОГРАММИРОВАНИЕ

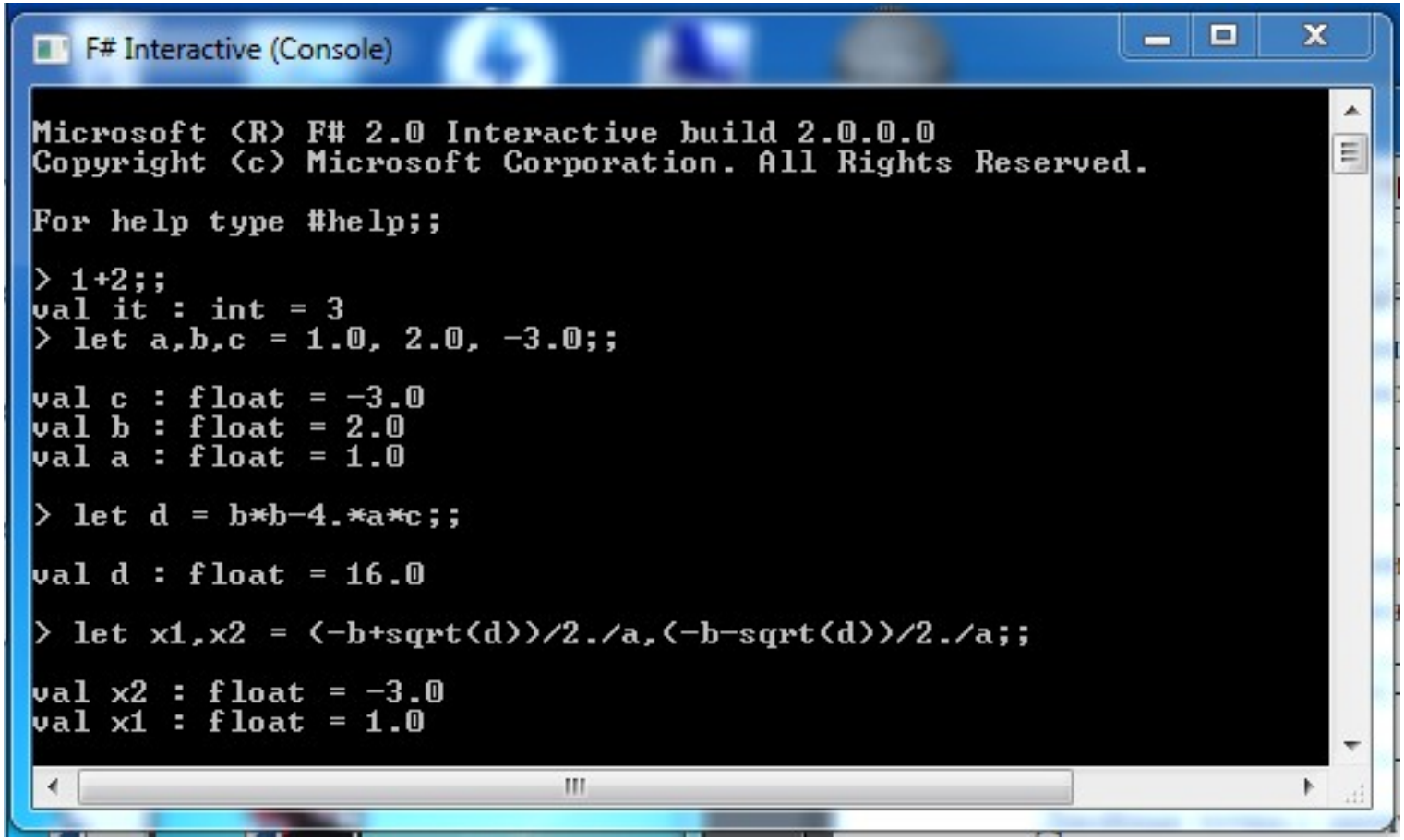
Лекция 6. Функциональное программирование на F#

В режиме интерпретации F# позволяет вам ввести выражение, которое затем он пытается вычислить и выдать результат. Вот пример простого диалога, вычисляющего арифметическое выражение:

```
> 1+2;;
val it : int = 3
```

Здесь жирным выделен текст, вводимый пользователем (после знака >), остальное – приглашение и ответ интерпретатора. Вот чуть более сложный пример решения квадратного уравнения:

```
> let a,b,c = 1.0, 2.0, -3.0;;
val c : float = -3.0
val b : float = 2.0
val a : float = 1.0
> let d = b*b-4.*a*c;;
val d : float = 16.0
> let x1,x2 = (-b+sqrt(d))/2./a,(-b-sqrt(d))/2./a;;
val x2 : float = -3.0
val x1 : float = 1.0
```


F# Interactive (Console)
Microsoft (R) F# 2.0 Interactive build 2.0.0.0
Copyright (c) Microsoft Corporation. All Rights Reserved.
For help type #help;;
> 1+2;;
val it : int = 3
> let a,b,c = 1.0, 2.0, -3.0;;
val c : float = -3.0
val b : float = 2.0
val a : float = 1.0
> let d = b*b-4.*a*c;;
val d : float = 16.0
> let x1,x2 = (-b+sqrt(d))/2./a,(-b-sqrt(d))/2./a;;
val x2 : float = -3.0
val x1 : float = 1.0

Представим себе математика, которому нужно решить некоторую задачу. Обычно задача формулируется как необходимость вычислить некоторый результат по имеющимся входным данным. В самом простейшем случае такое вычисление может задаваться обычным арифметическим выражением, например для нахождения одного корня квадратного уравнения $x^2 + 2x - 3$ существует явная формула, которую можно записать на F# следующим образом:

```
(-2.0+sqrt(2.0*2.0-4.0*(-3.0))) / 2.0 ;;
```

Если такое выражение ввести в ответ на приглашение интерпретатора F#, то мы получим искомый результат:

```
> (-2.0+sqrt(2.0*2.0-4.0*(-3.0))) / 2.0 ;;
val it : float = 1.0
```

Двойная точка с запятой в конце свидетельствует о том, что набранный текст можно передавать на исполнение интерпретатору. Отдельные же выражения можно разделять точкой с запятой или переходом на новую строку.

Обычно, конечно, задача не может быть решена одним лишь выражением. На деле при рассмотрении решения квадратного уравнения сначала вычисляют дискриминант D = $b^2 - 4ac$ (используя для его обозначения некоторую букву или **имя**, D) и затем уже – сами корни. В математических терминах пишут:

$x_1 = (-b + \sqrt{D})/(2a)$, где $D = b^2 - 4ac$,

или

пусть $D = b^2 - 4ac$, тогда $x_1 = (-b + \sqrt{D})/(2a)$.

На языке F# соответствующая запись примет следующий вид:

```
let D = 2.0*2.0-4.0*(-3.0) in (-2.0+sqrt(D)) / 2.0 ;;
```

Здесь let обозначает введение именованного обозначения – в следующем за in выражении буква D будет обозначать соответствующую формулу. Изменить значение D (в той же области видимости) уже невозможно.

С использованием let можно описать решение уравнения следующим образом:

```
let a = 1.0 in
 let b = 2.0 in
  let c = -3.0 in
   let D = b*b-4.*a*c in
    (-b+sqrt(D)) / (2.*a) ;;
```

Приведенный выше пример позволял нам вычислить лишь один корень квадратного уравнения. Для вычисления второго корня при таком подходе нам пришлось бы выписать аналогичное выражение, заменив в одном месте «+» на «−». Безусловно, такое дублирование кода не является допустимым.

В данном случае проблема легко решается использованием *пары значений*, или, более строго, *упорядоченного кортежа* (tuple) как результата вычислений. Упорядоченный набор значений является базовым элементом функционального языка, и с его использованием выражение для нахождения обоих корней уравнения запишется так:

```
let a = 1.0 in
 let b = 2.0 in
  let c = -3.0 in
   let D = b*b-4.*a*c in
    ((-b+sqrt(D))/(2.*a),(-b-sqrt(D))/(2.*a)) ;;
```

В результате мы получим такой ответ системы:

```
val it : float * float = (1.0, -3.0)
```

Здесь `it` – это специальная переменная, содержащая в себе результат последнего вычисленного выражения, а `float*float` – тип данных результата, в данном случае декартово произведение float на float, то есть пара значений вещественного типа.

Мы видим, что компилятор способен самостоятельно определить тип выражения – это называется *автоматическим выводом типов*. Вывод типов – это одна из причин, по которой программы на функциональных языках выглядят так компактно – ведь практически никогда не приходится в явном виде указывать типы значений для вновь описываемых имен и функций.

Помимо упорядоченных кортежей, F# содержит также встроенный синтаксис для работы со *списками* – последовательностями значений одного типа. Мы могли бы вернуть список решений (вместо пары решений), используя следующий синтаксис:

```
let a = 1.0 in
 let b = 2.0 in
  let c = -3.0 in
   let D = b*b-4.*a*c in
    [(-b+sqrt(D))/(2.*a);(-b-sqrt(D))/(2.*a)];;
```

Результат в этом случае выглядел бы так:

```
val it : float list = [1.0; -3.0]
```

Здесь `float list` – это список значений типа `float`. Суффикс `list` применим к любому типу и представляет собой описание полиморфного типа данных. Под-

Операция решения квадратного уравнения является достаточно типовой и вполне может пригодиться нам в дальнейшем при написании довольно сложной программы. Поэтому было бы естественно иметь возможность описать процесс решения квадратного уравнения как самостоятельную функцию. Наверное, вы уже догадались, что на вход она будет принимать тройку аргументов – коэффициенты уравнения, а на выходе генерировать пару чисел – два корня. Описание функции и ее применение для решения конкретного уравнения будут выглядеть следующим образом:

```
let solve (a,b,c) =
   let D = b*b-4.*a*c in
    ((-b+sqrt(D))/(2.*a),(-b-sqrt(D))/(2.*a))
 in solve (1.0,2.0,-3.0);;
```

Здесь сначала определяется функция solve, внутри нее определяется локальное имя D (локальное – это значит, что вне функции оно недоступно), а затем эта функция применяется для решения исходного уравнения с коэффициентами 1, 2 и –3.

Обратите внимание, что для описания функции используется тот же самый оператор `let`, что и для определения имен. На самом деле в функциональном программировании функции являются базовым типом данных (как еще говорят – *first-class citizens*), и вообще говоря, различия между данными и функциями делаются минимальные[1]. В частности, функции можно передавать в качестве параметров другим функциям и возвращать в качестве результата, можно описывать функциональные константы и т. д.

Для удобства в F# применяется также специальный синтаксис (так называемый #light-синтаксис), в котором можно опускать конструкцию `in`, просто записывая описания функций последовательно друг за другом. Вложение конструкций в этом случае будет определяться отступами – если выражение записано с отступом по сравнению с предыдущей строчкой, то оно является вложенным по отношению к нему, локальным. В таком синтаксисе приведенный пример запишется так:

```
let solve (a,b,c) =
   let D = b*b-4.*a*c
   ((-b+sqrt(D))/(2.*a),(-b-sqrt(D))/(2.*a));
solve (1.0,2.0,-3.0);;
```

Интересно посмотреть, какой тип данных в этом случае будет иметь функция `solve`. Как вы, наверное, догадались, она отображает тройки значений `float` в пары решений, что в нашем случае запишется как `float*float*float -> float*float`. Стрелка означает так называемый *функциональный тип* – то есть функцию, отображающую одно множество значений в другое.

В F# также существует конструкция для описания константы функционального типа, или так называемое *лямбда-выражение*. Свое название оно получило от лямбда-исчисления, математической теории, лежащей в основе функционального программирования. В лямбда-исчислении, чтобы описать функцию, вычисляющую выражение $x^2 + 1$, используется нотация $\lambda x.x^2 + 1$. Аналогичная запись на F# будет выглядеть так:

```
fun x -> x*x+1
function x -> x*x+1
```

В данном случае обе эти записи эквивалентны, хотя в будущем мы расскажем о некоторых различиях между `fun` и `function`. С использованием приведенной нотации наш пример можно также переписать следующим образом:

```
let solve = fun (a,b,c) ->
   let D = b*b-4.*a*c
   ((-b+sqrt(D))/(2.*a),(-b-sqrt(D))/(2.*a));
```

Часто, как в нашем прошлом примере, бывает необходимо описать функцию с несколькими аргументами. В лямбда-исчислении и в функциональном программировании мы всегда оперируем функциями от одного аргумента, который, однако, может иметь сложную природу. Как в прошлом примере, всегда можно передать в функцию в качестве аргумента кортеж, тем самым передав множество значений входных параметров.

Однако в функциональном программировании распространен и другой прием, называемый *каррированием*. Рассмотрим функцию от двух аргументов, например сложение. Ее можно описать на F# двумя способами:

```
let plus (x,y) = x+y
let cplus x y = x+y
```

Первый случай похож на рассмотренный ранее пример, и функция `plus` будет иметь тип `int*int -> int`. Второй случай – это как раз каррированное описание функции, и cplus будет иметь тип `int -> int -> int`, что на самом деле, используя соглашение о расстановке скобок в записи функционального типа, означает `int -> (int -> int)`.

Смысл каррированного описания – в том, что функция сложения применяется к своим аргументам «по очереди». Предположим, нам надо вычислить `cplus 1 2` (применение функции к аргументам в F# записывается как и в лямбда-исчислении, без скобок, простым указанием аргументов вслед за именем функции). Применяя cplus к первому аргументу, мы получаем значение функционального типа `int->int` – функцию, которая прибавляет единицу к своему аргументу. Применяя затем эту функцию к числу 2, мы получаем искомый результат 3 – целого типа. Запись plus 1 2, таким образом, рассматривается как (plus 1) 2, то есть сначала мы

получим функцию инкремента, а потом применим ее к числу 2, получив требуемый результат. В частности, все стандартные операции могут быть использованы в каррированной форме путем заключения операции в скобки и использования префиксной записи, например:

```
(+) 1 2;;
let incr = (+)1;;
```

В нашем примере с квадратным уравнением мы также могли бы описать каррированный вариант функции solve:

```
let solve a b c =
   let D = b*b-4.*a*c
   ((-b+sqrt(D))/(2.*a),(-b-sqrt(D))/(2.*a));;
solve 1.0 2.0 -3.0;;
```

Такой подход имеет как минимум одно преимущество – с его помощью можно легко описать функцию решения линейных уравнений как частный случай решения квадратных при a = 0:

```
let solve_lin = solve 0.0;;
```

Правда, вдумчивый читатель сразу заметит, что наша функция решения не предназначена для использования в ситуациях, когда a = 0, – в этом случае будет происходить деление на 0. Как расширить функцию solve для правильной обработки различных ситуаций, мы узнаем в следующем разделе.

Поскольку такая ситуация возникает достаточно часто, то соответствующий тип данных присутствует в языке и называется *опциональным типом* (option type). Например, значения типа `int option` могут содержать в себе либо конструкцию Some(...) от некоторого целого числа, либо специальную константу None. В нашем случае функция решения уравнения, возвращающая опциональный тип, будет описываться так:

```
let solve a b c =
   let D = b*b-4.*a*c
   if D<0. then None
   else Some(((-b+sqrt(D))/(2.*a),(-b-sqrt(D))/(2.*a)));;
```

Сама функция в этом случае будет иметь тип `solve : float -> float -> float -> (float * float) option` – этот тип будет выведен компилятором автоматически. Работать с опциональным типом можно примерно следующим образом:

```
let res = solve 1.0 2.0 3.0 in
 if res = None
   then "Нет решений"
   else Option.get(res).ToString();;
```

На самом деле опциональный тип представляет собой частный случай типа данных, называемого *размеченным объединением* (discriminated union). Он мог бы быть описан на F# следующим образом:

```
type 'a option = Some of 'a | None
```

В нашем примере, чтобы описать более общий случай решения как квадратных, так и линейных уравнений, мы опишем решение в виде объединения трех различных случаев: отсутствие решений, два корня квадратного уравнения и один корень линейного уравнения:

```
type SolveResult =
   None
 | Linear of float
 | Quadratic of float*float
```

В данном случае мы описываем тип данных, который может содержать либо значение `None`, либо `Linear(...)` с одним аргументом типа `float`, либо `Quadratic(...)` с двумя аргументами. Сама функция решения уравнения в общем случае будет иметь такой вид:

```
let solve a b c =
   let D = b*b-4.*a*c
   if a=0. then
     if b=0. then None
     else Linear(-c/b)
   else
     if D<0. then None
     else Quadratic(((-b+sqrt(D))/(2.*a),(-b-sqrt(D))/(2.*a)))
```

Для определения того, какой именно результат вернула функция `solve`, необходимо воспользоваться специальной конструкцией *сопоставления с образцом* (pattern matching):

```
let res = solve 1.0 2.0 3.0
match res with
  None -> printf "Нет решений"
| Linear(x) -> printf "Линейное уравнение, корень: %f" x
| Quadratic(x1,x2) -> printf "Квадратное уравнение, корни: %f %f" x1 x2
```

Операция match осуществляет последовательное сопоставление значения выражения с указанными шаблонами, при этом при первом совпадении вычисляется и возвращается соответствующее выражение, указанное после стрелки. В процессе сопоставления также происходит связывание имен переменных в шаблоне с соответствующими значениями. Возможно также указание более сложных условных выражений после шаблона, например:

```
match res with
  None -> printf "Нет решений"
| Linear(x) -> printf "Линейное уравнение, корень: %f" x
| Quadratic(x1,x2) when x1=x2 -> printf "Квадр.уравнение,1 корень: %f" x1
| Quadratic(x1,x2) -> printf "Квадр. уравнение,2 корня: %f %f" x1 x2
```

Следует отметить, что сопоставление с образцом в F# может производиться не только в рамках конструкции `match`, но и при сопоставлении имен `let` и при описании функциональной константы с помощью ключевого слова `function`. В частности, следующие два описания функции получения текстового результата решения уравнения text_res эквивалентны:

```
let text_res x = match x with
  None -> "Нет решений"
| Linear(x) -> "Линейное уравнение, корень: "+x.ToString()
| Quadratic(x1,x2) when x1=x2 ->
"Квад.уравнение, один корень: "+x1.ToString()
| Quadratic(x1,x2) ->
"Квадратное уравнение, два корня:"+x1.ToString()+x2.ToString()

let text_res = function
  None -> "Нет решений"
| Linear(x) -> "Линейное уравнение, корень: "+x.ToString()
| Quadratic(x1,x2) when x1=x2 ->
"Квадратное уравнение, один корень: "+x1.ToString()
| Quadratic(x1,x2) ->
"Квадратное уравнение, два корня:"+x1.ToString()+x2.ToString()
```

Наиболее часто распространенным примером использования конструкции сопоставления с образцом внутри let является одновременное сопоставление нескольких имен, например:

```
let x1,x2 = -b+sqrt(D))/(2.*a),(-b-sqrt(D))/(2.*a)
```

В данном случае на самом деле происходит сопоставление одной упорядоченной пары типа `float` с другой упорядоченной парой, что приводит к попарному сопоставлению обоих имен.

В любом языке программирования одна из важнейших задач – выполнение повторяющихся действий. В императивных языках программирования для этого используются циклы (с предусловием, со счетчиком и т. д.), однако циклы основаны на изменении так называемого *инварианта цикла* и поэтому не могут быть использованы в функциональном подходе. Здесь нам на помощь приходит понятие *рекурсии*.

Например, рассмотрим простейшую задачу – печать всех целых чисел от A до B. Для решения задачи при помощи рекурсии мы думаем, как на каждом шаге выполнить одно действие (печать первого числа, A), после чего свести задачу к применению такой же функции (печать всех чисел от A + 1 до B). В данном случае получится такое решение:

```
let rec print_ab A B =
  if A>=B then printf "%d " A
  else
    printf "%d " A
    print_ab (A+1) B
```

Здесь ключевое слово rec указывает на то, что производится описание рекурсивных действий. Это позволяет правильно проинтерпретировать ссылку на функцию с тем же именем print_ab, расположенную в правой части определения. Без ключевого слова rec компилятор пытался бы найти имя print_ab, определенное в более высокой области видимости, и связать новое имя print_ab с другим выражением для более узкого фрагмента кода.

Очевидно, что решать каждый раз задачу выполнения повторяющихся действий с помощью рекурсии, описывая отдельную функцию, неудобно. Поэтому мы можем выделить идею итерации как отдельную абстракцию, а выполняемое действие передавать в функцию в качестве параметра. В этом случае мы получим следующее описание функции итерации:

```
let rec for_loop f A B =
  if A>=B then f A
  else
    f A
    for_loop f (A+1) B
```

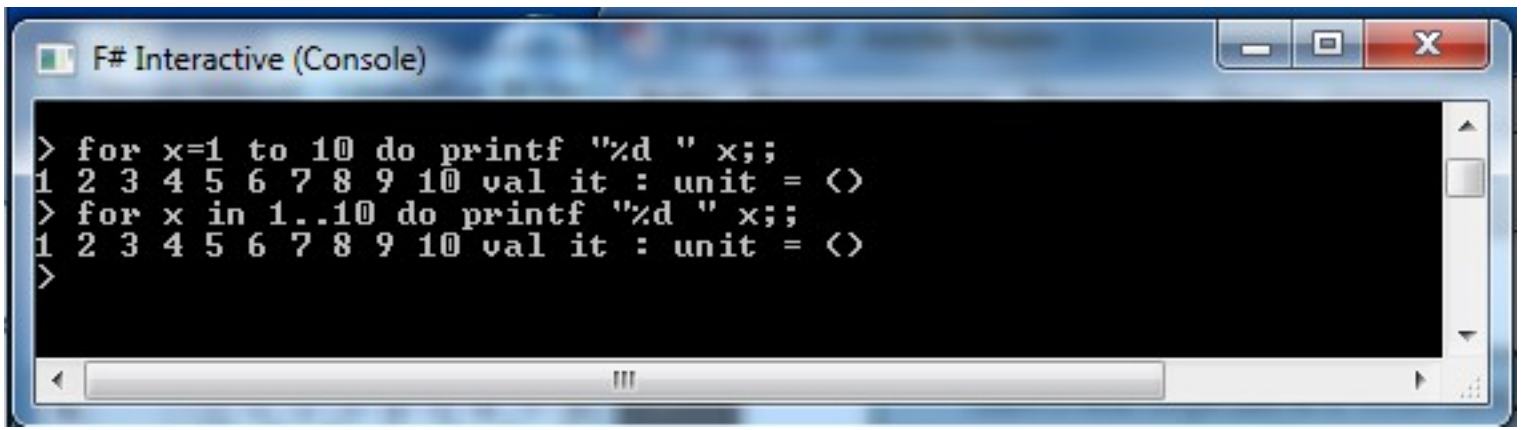
Такое абстрактное описание понятия итерации мы теперь можем применить для печати значений от 1 до 10 следующим образом:

```
for_loop (fun x -> printf "%d " x) 1 10
```

Здесь мы передаем в тело цикла функциональную константу, описанную здесь же при помощи лямбда-выражения. В результате получившаяся конструкция напоминает обычный цикл со счетчиком, однако важно понимать отличия: здесь тело цикла представляет собой функцию, вызываемую с различными последовательными значениями счетчика, что, например, исключает возможную модификацию счетчика внутри тела цикла.

На самом деле цикл со счетчиком в такой интерпретации достаточно часто используется, поэтому в F# для его реализации есть специальная встроенная конструкция `for`. Например, для печати чисел от 1 до 10 ее можно использовать следующим образом:

```
for x=1 to 10 do printf "%d " x
for x in 1..10 do printf "%d " x
```



В качестве еще одного примера использования рекурсии рассмотрим определение функции `rpt`, которая будет возводить заданную функцию $f(x)$ в указанную степень n, то есть строить вычисление n-кратного применения функции f к аргументу x:

$$rpt\ n\ f\ x = f(f(...f(x)..)$$

Для описания такой функции вспомним, что $f^0(x) = x$ и $f^n(x) = f(f^{n-1}(x))$, тогда рекурсивное определение получится естественным образом:

```
let rec rpt n f x =
    if n=0 then x
    else f (rpt (n-1) f x)
```

В приведенном выше определении мы рассматривали функцию `rpt` применительно к некоторому аргументу x. Однако мы могли рассуждать в терминах функций, не опускаясь до уровня применения функции к конкретному аргументу. Заметим, что исходное рекуррентное определение можно записать так:

$f^0 = Id$

$f^n = f \circ f^{n-1}$

Здесь Id обозначает тождественную функцию, а знак $\circ$ – композицию функций. Такое рекуррентное соотношение на F# может быть записано следующим образом:

```
let rec rpt n f =
    if n=0 then fun x->x
    else f >> (rpt (n-1) f)
```

В этом определении знак >> описывает композицию функций. Хотя эта операция является встроенной в библиотеку F#, она может быть определена следующим образом:

```
let (>>) f g x = f(g x)
```

Помимо композиции, есть еще одна аналогичная конструкция |>, которая называется *конвейером* (pipeline) и определяется следующим образом:

```
let (|>) x f = f x
```

С помощью конвейера можно последовательно передавать результаты вычисления одной функции на вход другой, например (возвращаясь к решению квадратного уравнения):

```
solve 1.0 2.0 3.0 |> text_res |> System.Console.Write
```

В этом случае результат решения типа `SolveResult` подается на вход функции `text_res`, которая преобразует его в строку, выводимую на экран системным вызовом `Console.Write`. Такой же пример мог бы быть записан без использования конвейера следующим образом:

```
System.Console.Write(text_res(solve 1.0 2.0 3.0))
```

Очевидно, что в случае последовательного применения значительного количества функций синтаксис конвейера оказывается более удобным. Следует отметить, что в F# для удобства также предусмотрены обратные операторы конвейера <| и композиции <<.

1.9. Пример – построение множества Мандельброта

В качестве примера использования всех изученных конструкций F# рассмотрим более сложную задачу – построение фрактального изображения, знаменитого множества Мандельброта. Математически это множество определяется следующим образом: рассмотрим последовательность комплексных чисел $z_{n+1} = z_n^2 + c$, $z_0 = 0$. Для различных c эта последовательность либо сходится, либо расходится. Например, для $c = 0$ все элементы последовательности $z_i = 0$, а для $c = 2$ имеем расходящуюся последовательность. Множество Мандельброта – это множество тех c, для которых последовательность сходится.

Приступим к реализации алгоритма построения на F#. Для начала определим функцию mandelf, описывающую последовательность $z^2 + c$, – при этом необходимо в явном виде указать для аргументов тип `Complex`[1], поскольку по умолчанию для операции + полагается целый тип. Кроме того, чтобы тип Complex стал доступен, вначале придется указать преамбулу, открывающую соответствующие модули:

```
open System
open Microsoft.FSharp.Math

let mandelf (c:Complex) (z:Complex) = z*z+c
```

Следующим этапом определим функцию `ismandel: Complex->bool`, которая будет по любой точке комплексной плоскости выдавать признак ее принадлежности множеству Мандельброта. Для простоты мы будем рассматривать слегка видоизмененное множество, похожее на множество Мандельброта – множество тех точек, для которых $z_{20}(0)$ является ограниченной величиной, то есть по модулю меньше 1.

Для вычисления $z_{20}(0)$ вспомним, что функция `mandelf` описана в каррированном представлении и при некотором фиксированном c представляет собой функцию из `Complex` в `Complex`. Таким образом, используя описанную ранее функцию n-кратного применения функции `rpt`, мы можем построить 20-кратное применение функции `mandelf`: `rpt 20 (mandelf c)`. Далее остается применить эту функцию к нулю и взять модуль значения:

```
let ismandel c = Complex.Abs(rpt 20 (mandelf c) (Complex.zero))<1.0
```

По сути дела, эти две строчки – описание функций `mandelf` и `ismandel` – определяют нам множество Мандельброта. Построить это множество – для начала в виде рисунка из звездочек на консоле – теперь дело техники и нескольких строк кода:

```
let scale (x:float,y:float) (u,v) n = float(n-u)/float(v-u)*(y-x)+x;;

for i=1 to 60 do
 for j=1 to 60 do
   let lscale = scale (-1.2,1.2) (1,60) in
   let t = complex (lscale j) (lscale i) in
   Console.Write(if ismandel t then "*" else " ")
 Console.WriteLine("")
```

Результат работы программы в консольном режиме можно наблюдать на рис. 1.1. Для получения такого результата мы преобразовали программу в самостоятельное F#-приложение – файл с расширением `.fs`, который затем можно откомпилировать из Visual Studio либо с помощью утилиты `fsc.exe` в независимое выполняемое приложение.

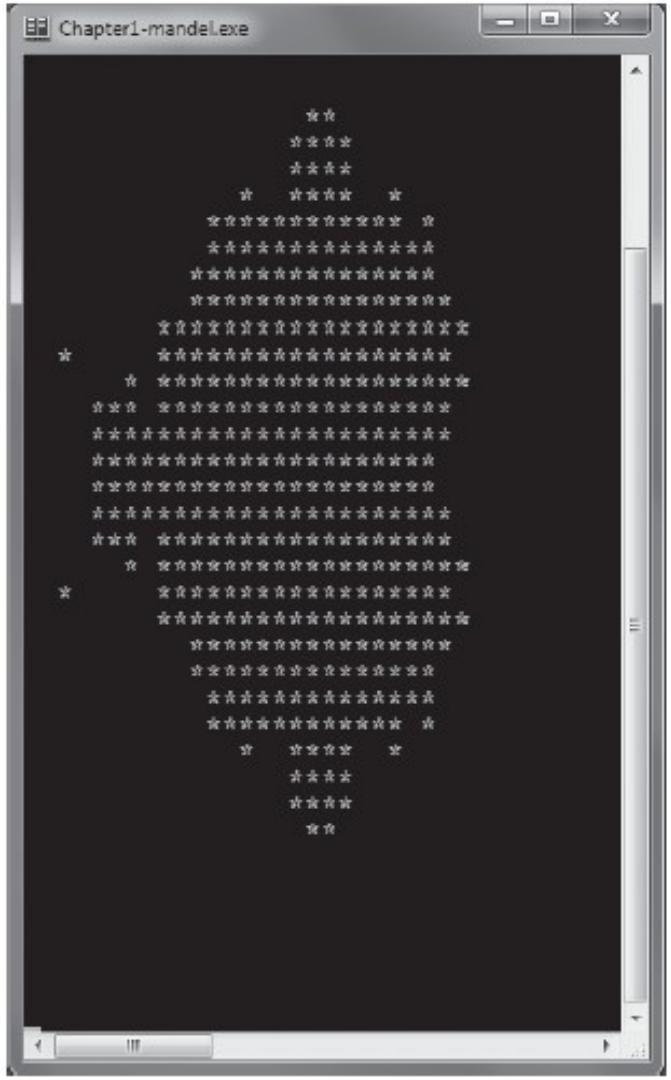


Рис. 1.1

Таким образом, программа, отвечающая за построение множества Мандельброта, уместилась, по сути дела, на одном экране компактного кода. Если проанализировать причины, по которым программа получилась существенно компактнее возможных аналогов на C#, можно отметить следующее:

- компактный синтаксис для описания функций;
- вывод типов, благодаря которому не надо практически нигде указывать тип данных используемых значений. Обратите внимание, что при этом язык остается *статически типизируемым*, то есть проверка типов производится на этапе компиляции программы!
- использование каррированного вызова функций, благодаря чему очень просто можно оперировать понятием частичного применения функции;
- наличие удобных встроенных типов данных для упорядоченных кортежей и списков.

Безусловно, построение множества Мандельброта из звездочек впечатляет, но было бы еще интереснее построить графическое изображение с большим разрешением. К счастью, F# является полноценным языком семейства .NET и может использоваться совместно со всеми стандартными библиотеками .NET, такими как System.Drawing для манипулирования двумерными изображениями и даже библиотекой Windows Forms.

Код, строящий фрактальное изображение в отдельном окне, приведен ниже:

```
open System.Drawing
open System.Windows.Forms

let form =
   let image = new Bitmap(400, 400)
   let lscale = scale (-1.2,1.2) (0,image.Height-1)
   for i = 0 to (image.Height-1) do
     for j = 0 to (image.Width-1) do
       let t = complex (lscale i) (lscale j) in
       image.SetPixel(i,j,
if ismandel t then Color.Black else Color.White)
   let temp = new Form()
   temp.Paint.Add(fun e -> e.Graphics.DrawImage(image, 0, 0))
   temp.Show()
   temp
```

В начале программы мы подключаем библиотеки Windows Forms и System. Drawing. Основная функция – form – отвечает за создание основного окна с фрактальным изображением. В этой функции сначала создается объект `Bitmap` – двумерный пиксельный массив, который заполняется фрактальным изображением с помощью двойного цикла, похожего на использованные в предыдущем примере. После заполнения изображения создается форма и добавляется для нее функция перерисовки, которая при каждой перерисовке окна отрисовывает внутри единожды вычисленное фрактальное изображение (рис. 1.2).

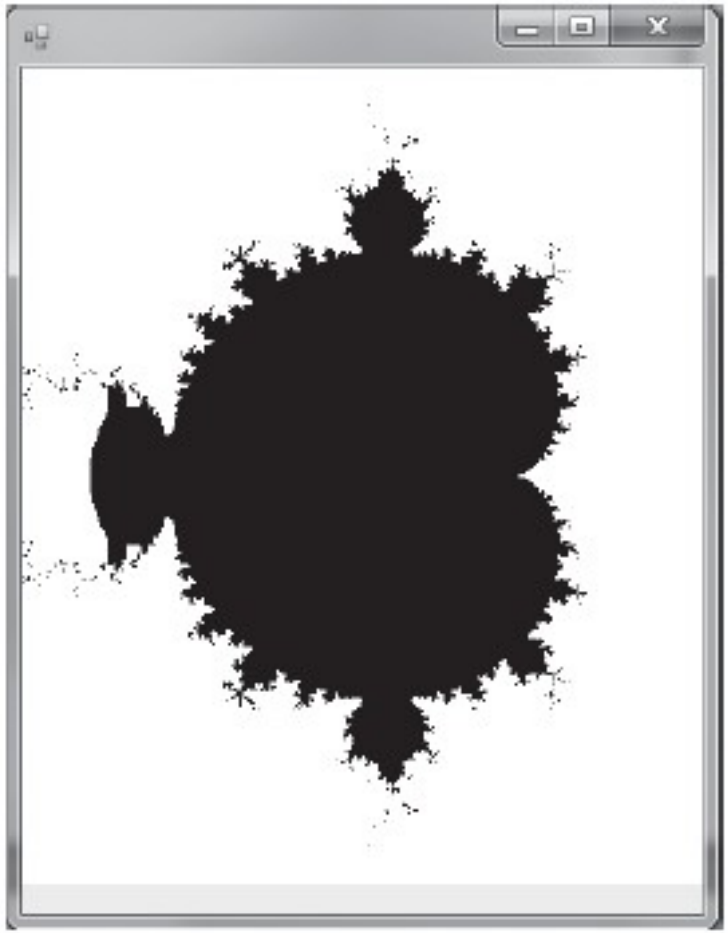


Рис. 1.2

Конечно, программа в таком виде имеет недостаточно богатый интерфейс, да и процесс построения формы через переопределение функции перерисовки не является самым правильным. Основная цель данного примера – показать, что F# может прозрачным образом работать со всем имеющимся многообразием функций платформы .NET, от графических примитивов до сетевого взаимодействия, от доступа к СУБД до построения Silverlight-приложений.

Важно, однако, понимать, что F# не является заменой традиционным языкам типа C# и Visual Basic. Предполагается, что для построения интерфейсов приложений с использованием визуальных дизайнеров будут по-прежнему использоваться императивные языки, а F# сможет эффективно применяться для решения задач, связанных с обработкой данных. Грамотное разделение кода и используемого языка программирования между отдельными модулями – это необходимое условие успешности и эффективности разработки программного проекта.