УДК 004.432.42

Принципы функционального программирования

Л. А. Залогова
Пермский государственный национальный исследовательский университет
Россия, 614990, г. Пермь, ул. Букирева, 15
zalogova.la@gmail.com

В последнее время наблюдается рост интереса к функциональному программированию. В отличие от императивного программирования функциональная парадигма представляет собой другой подход к разработке программ, который является более эффективным для решения некоторого класса задач. В статье рассмотрены и систематизированы принципы, характерные для различных функциональных языков. Статья адресована тем, кто владеет навыками императивного программирования и желает познакомиться с концепциями функциональных языков. Для демонстрации примеров использован процедурный язык Паскаль и функциональный язык F#.

Ключевые слова: *чистые функции; композиции функций; неизменность данных; функции высших порядков; хвостовая рекурсия.*

DOI: 10.17072/1993-0550-2020-2-54-68

1. Императивный и функциональный стиль программирования[1]

Определенный способ мышления (парадигма) служит основой для создания языка программирования. Существуют разные парадигмы программирования – процедурная, объектно-ориентированная, функциональная, логическая. Каждая из парадигм используется для решения определенного класса задач. Некоторые языки поддерживают несколько парадигм, другие же, наоборот, ориентированы на реализацию только одной парадигмы. Для каждого языка программирования одна из парадигм является основной, а другие парадигмы – дополнительными.

1.1. Императивный стиль программирования

Основными понятиями программ, написанных на машинно-ориентированных, процедурных и объектно-ориентированных языках, являются: переменная (ячейка памяти), инструкция (команда или оператор), состояние.

Переменная – поименованное место памяти, куда можно записать значение. Значение переменной можно читать столько раз, сколько пожелаете. Его можно стереть и на его место записать новое значение. Таким образом, переменные обладают значениями, которые могут изменяться в процессе исполнения программы.

Инструкция определяет некоторое действие, например присваивание, ввод, вывод, ветвление, цикл и др.

Состояние – значения некоторого множества (набора) переменных.

Программы, написанные на машинно-ориентированных, процедурных и объектно-ориентированных языках, основаны на *изменении состояний* (значений переменных). Входные данные программы представляют собой *первоначальное состояние* S. В результате выполнения инструкций происходит последовательное изменение состояний:

$$S \rightarrow S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow \ldots \rightarrow S_N,$$

т.е. создаются *промежуточные состояния* S_1, S_2, S_3, …, S_{N-1}. Выходные же данные (результаты) являются *заключительным состоянием* S_N.

Например, в программе сортировки первоначальное состояние – массив, который нужно отсортировать, промежуточные состояния –

значения элементов массива на очередном этапе сортировки, заключительное состояние – отсортированный массив. Такой стиль программирования называют *императивным* (от лат. *imperatives* – повелительный).

Императивное программирование — это парадигма программирования, которая описывает процесс вычислений в виде инструкций, изменяющих состояния. Обычно освоение программирования начинается с изучения языка Паскаль или С. Поэтому для начинающих программистов первой парадигмой является процедурная, т.е. императивная.

Императивный стиль основан на особенностях архитектуры компьютеров, которая была предложена Джоном фон Нейманом в 50-е годы прошлого века и используется в настоящее время. Согласно архитектуре Д. фон Неймана основными компонентами компьютера являются *память* (пронумерованные ячейки), содержащая программу и данные; *центральный процессор*, выполняющий команды по обработке данных; а также *внешние устройства*. Процесс обработки информации осуществляется в соответствии с заранее составленной программой. Процессор выполняет инструкции (команды) программы, которые изменяют последовательные состояния памяти. Императивное программирование учитывает принципы организации архитектуры компьютера. Однако существуют и другие стили программирования, например функциональное программирование.

1.2. Функциональный стиль программирования

В функциональном программировании программа описывается с использованием *математических функций.* Функциональная программа имеет сравнительно простую форму: последовательность определений функций, за которой следует последовательность вызовов этих функций. Вычисления начинаются с вызова некоторой функции, которая, в свою очередь, вызывает другие функции и т.д. Каждый вызов возвращает некоторое значение в вызывающую функцию, вычисление которой после этого продолжается. Этот процесс повторяется до тех пор, пока запустившая вычисления функция не вернет конечный результат. Например, используя функцию `макс2` для нахождения наибольшего из двух чисел, наибольшее из четырех чисел можно определить так: `макс2(макс2(a,в), макс2(c,d))`

Функции только принимают входные данные и возвращают некоторый результат. Взаимодействие между разными функциями возможно только через вызовы во время выполнения, при этом порядок вычисления подвыражений не имеет значения. Результат выполнения функциональной программы – значение выражения, которое определяется в терминах базовых и/или определенных пользователем функций.

При работе с функциональной парадигмой важно постоянно думать в терминах функций. Составлять программы с помощью функций можно и на императивных языках, однако функциональный подход имеет существенные отличия (см. раздел 3).

Функциональная парадигма не должна рассматриваться как замена императивному программированию. Она лишь представляет другой подход к разработке программ. Для решения некоторого класса задач такой подход является более эффективным.

В последнее время наблюдается рост интереса к функциональному программированию. Все больше людей интересуются методологией и технологией построения программ с использованием функциональной парадигмы. Эта парадигма наиболее приспособлена и применяется для создания систем искусственного интеллекта, разработки графических интерфейсов, создания графических приложений реального времени, разработки систем параллельных вычислений, построения компиляторов. Кроме того, удобство функционального программирования отмечают специалисты, работающие в финансовой и научной областях, а также занимающиеся техническими расчетами.

2. Концепция λ-исчисления

Функциональное программирование основано на математической системе, которая называется λ-исчисление.

λ-исчисление было создано для того, чтобы исследовать задачи, связанные с вычислениями. В 1930-х гг. американский математик Алонзо Черч разработал систему для формализации понятия "вычисление" и назвал ее *λ-исчисление*. Рассмотрим лишь основные его аспекты, применяемые в функциональном программировании.

2.1. Понятие λ-функции

Основу λ-исчисления составляют λ-функции.

λ-функция – анонимная функция одного аргумента (функция без имени – уникального идентификатора для доступа).
λ-абстракция – определение анонимной функции.

Пример 2.1. Определение λ-функции увеличения аргумента на 2:

```
λx.x+2.
```

Символ λ означает "отобразить", аргумент функции – x, а тело функции – x+2. Аргумент отделяется от тела символом "точка".

В императивных языках функции сначала определяются, а потом вызываются, как правило, в различных точках программы. При вызове указывается имя и фактические параметры функции. λ-функция не имеет имени, поэтому она определяется и *сразу же применяется* к аргументу (в λ-исчислении процесс передачи функции фактического параметра называется *применением*).

Пример 2.2. Определение и применение λ-функции увеличения аргумента на 10:

```
λx.x+10 3.
```

В случае применения этой функции вхождение аргумента x в теле функции заменяется значением 3. В результате вычисления получаем 13.

Таким образом, абстракция и применение – основные операции λ-исчисления.

2.2. Функции – это данные

В λ-исчислении нет различий между данными и функциями. Это означает, что функциями можно оперировать как данными, т.е. передавать их в качестве аргументов и получать в качестве результатов других функций.

Пример 2.3. Результат λ-функции – функция

```
λx.λy.y+x+10 5.
```

Здесь λ-функция принимает аргумент x=5 и возвращает функцию λy.y+15.

2.3. Каррирование

В λ-исчислении отсутствует встроенная поддержка функций с несколькими аргументами. В связи с этим работа с функцией от нескольких аргументов сводится к работе с функциями от одного аргумента. Это означает, что все функции с несколькими аргументами в процессе применения проходят процесс *каррирования.* Этот процесс заключается в следующем: функция с n (n>1) аргументами сначала применяется с первым аргументом и возвращает новую функцию с (n-1) аргументом. Эта новая функция немедленно применяется со своим первым аргументом (это второй аргумент исходной функции) и возвращает функцию с (n-2) аргументами. Процесс продолжается до тех пор, пока не будут применены функции с (n-3), (n-4), ….1 аргументом, и, таким образом, получен окончательный результат.

Пример 2.4. Вычисление значения функции с двумя аргументами.

Вычисление значения функции (λx.λy.x+y)2 5 с двумя аргументами x и y выполняется в два этапа: получив значение 2 для первого аргумента, функция выдает новую функцию (λy.2+y)5, в результате применения которой возвращается число 7.

Пример 2.5. Вычисление значения функции с тремя аргументами.

Вычисление значения функции с тремя аргументами (λx.λy.λz.x+y+z)2 5 10 выполняется в три этапа: получив значение 2 для первого аргумента, функция выдает новую функцию от двух аргументов (λy.λz.2+y+z)5 10. Применение этой функции с аргументом 5 возвращает функцию (λz.7+z) 10, в результате применения которой получаем число 17.

3. Особенности функций

3.1. Чистые функции

Одним из свойств функций, рассматриваемых в функциональном программировании, является их чистота. Для определения чистых функций вводятся два понятия: детерминированная функция и побочный эффект.

Детерминированная функция – функция, которая всегда возвращает один и тот же результат для одинаковых аргументов. Для вычисления значения детерминированная функция использует только переданные ей аргументы.

Пример 3.1. Паскаль. Детерминированная функция

```
function determ1
   (x:integer,y:integer):integer;
begin determ1:= x+y end;
```

Значение функции determ1 определяется только ее аргументами. Результат же *недетерминированной функции* зависит не только от входных аргументов.

Пример 3.2. Паскаль. Недетерминированная функция

```
z=0;
```

```
function nondeterm1
   (x:integer,y:integer):integer;
begin nondeterm1:= z+x+y end;
```

Функция `nondeterm1` является недетерминированной, так как результат ее вызова зависит не только от аргументов `x` и `y`, но и от значения глобальной переменной `z`. В результате различных вызовов этой функции с одинаковыми аргументами будут получаться разные результаты, зависящие от значения `z`. При использовании такой функции могут возникнуть трудности, так как необходимо знать контекст ее вызова (значение глобальной переменной). Это, в свою очередь, требует рассмотрения истории вычислений.

Побочный эффект – то, что делает функция, кроме вычисления возвращаемого значения. Функция с побочными эффектами не только вычисляет значение на основе входных данных, но и выполняет дополнительные действия. Например, выводит результаты на экран, записывает данные в файл, изменяет таблицы баз данных, реагирует на исключительные ситуации и т.д. Детерминированные функции могут содержать побочные эффекты.

Пример 3.3. Паскаль. Детерминированная функция с побочным эффектом.

```
function determ2
   (x:integer,y:integer):integer;
begin writeln ("summa= ",x+y);
      determ2:= x+y
end;
```

Функция `determ2` содержит побочный эффект – печать суммы двух чисел, однако она является детерминированной, так как ее результирующее значение зависит только от входных аргументов `x` и `y`.

Императивные языки программирования содержат глобальные переменные. Изменение их значений в теле функции (например, оператором присваивания) является побочным эффектом. Результат такого побочного эффекта состоит в том, что при повторном вызове может измениться значение функции, даже если аргументы остались неизменными. Побочные эффекты не всегда приводят к плохим результатам, однако в непредусмотренных случаях могут быть источником ошибок.

Чистая функция – детерминированная функция без побочных эффектов. Чистая функция – это функция в математическом смысле.

Пример 3.4. Паскаль. Чистая функция.

```
function average3
        (x,y,z: integer):integer;
begin
   average3 := (x+y+z)/3
end;
```

Функция `average3` является чистой, так как ее результат определяется только входными параметрами, и она не выполняет никаких других действий, кроме вычисления результата – среднего значения трех чисел.

Таким образом, чистая функция, вызванная в разных контекстах для одинаковых значений входных параметров, всегда возвращает одно и то же значение и не имеет побочных эффектов. Значение чистой функции определяется только ее аргументами. Все чистые функции являются детерминированными, однако не все детерминированные функции – чистые. Например, если детерминированная функция выводит значения на принтер, то она не будет чистой. Чистые функции могут вызывать только чистые функции.

В функциональном программировании, как правило, используются чистые функции. Однако не удается полностью исключить побочные эффекты, так как программа должна взаимодействовать с внешним миром. В этом случае ставится цель – свести количество побочных эффектов к минимуму и отделить (изолировать) их от основной части программы. Например, все действия по выводу результатов обычно выносят в отдельную функцию, которая, в свою очередь, вызывает чистые функции.

Преимущества чистых функций

Значения чистых функций зависят только от параметров и не зависят от внешнего окружения, поэтому их проще отлаживать, тестировать, повторно использовать, из них легче составлять сложные программы. Стандартные математические библиотеки почти всех языков программирования, как правило, содержат только чистые функции.

Использование чистых функций – хороший стиль программирования.

Так как чистые функции не имеют побочных эффектов, то они могут вычисляться в произвольном порядке или одновременно. Это означает, что функциональные программы, состоящие из чистых функций, хорошо поддаются *распараллеливанию*, т.е. компьютер может вычислять значения разных функций на разных процессорах. Императивные же программы часто задают фиксированный

порядок вычислений, поэтому их обработка на нескольких процессорах усложняется.

3.2. Анонимные функции

В функциональном программировании, основанном на λ-исчислении, используются *анонимные функции*.

Пример 3.5. F#. Описание и немедленный вызов анонимной функции

```
let x = (fun x y -> x + y ) 1 2
```

Описание анонимной функции начинается с ключевого слова `fun`, за которым следует список аргументов `x y`; после знака `->` записывается тело функции; `1 2` – фактические параметры. В результате применения анонимной функции переменная `x` именует значение `3`.

Нет необходимости присваивать функции имя, если она передается в качестве параметра другой функции. В этом случае функцию-параметр удобно оформить в виде анонимной функции (см. раздел 9).

3.3. Каррирование

Важно помнить, что *каррирование* – это возможность функции принимать аргументы по одному, каждый раз возвращая новую функцию от меньшего числа аргументов. В функциональном программировании все функции с несколькими аргументами в процессе применения каррируются. Каррирование сохраняет математическую строгость в определении функции, даже несмотря на то, что используются функции многих переменных.

Пусть описана функция умножения двух целых чисел:

```
let mult x y = x*y
```

Можно каррировать (зафиксировать) первый аргумент функции и, таким образом, создать функцию, принимающую второй аргумент:

```
let multFive = mult 5
```

Значение 5 передается функции `mult` и в результате получается новая функция от одного аргумента `multFive`, которая прибавляет число 5 к своему аргументу. Один из вариантов применения каррированной функции `multFive`:

```
let result = multFive 10
```

Ответ – `15`.

Каррирование подтверждает идею о том, что функции есть значения: функция получает аргументы по одному и каждый раз в качестве значения возвращает новую функцию.

В традиционной математической записи аргументы вызова функции заключаются в круглые скобки. Такая же запись используется в императивных языках. В функциональных же языках для применения функции принято другое обозначение: имя функции отделяется от аргументов пробелом; кроме того, все аргументы разделяются между собой пробелами.

Пример 3.6. Использование каррирования для печати целых и вещественных чисел.

Функция `printfn` имеет два аргумента – строку формата и данные. Зафиксируем первый параметр (формат вывода), например, `"%d"`. В результате получим каррированную функцию, которая принимает число и выводит его на экран в формате целого:

```
let printInt = printfn "%d".
```

Применение функции `printInt`:

```
let x=3
printInt x
let y=5.12
printInt (int y).
```

Теперь определим и применим каррированную функцию для печати вещественных чисел:

```
let printfloat = printfn "\n %f"
let sq x = x*x
printfloat (float (sq 3))
```

Здесь каррирование позволило создать из функции с двумя аргументами несколько функций с одним аргументом. В общем случае сначала создается обобщенный вариант функции с несколькими аргументами, а затем используется каррирование для создания частных вариантов этой функции с меньшим числом аргументов. Каррировать аргументы можно только слева направо.

В функциональных языках все функции каррированы по умолчанию. Однако это не всегда удобно.

Например, если функция принимает в качестве параметра координаты точки, которые должны обрабатываться совместно, а не последовательно. В этом случае используется кортеж `(x,y)`, который обрабатывается как единый параметр (см. раздел 6). Использование кортежа как параметра функции исключает каррирование.

Еще одна особенность функциональных языков – частично применимые функции (ЧПФ). *Частичное применение* дает возможность зафиксировать значения нескольких

аргументов функции и создать новую функцию с меньшим числом аргументов.

Пример 3.7. F#. Частичное применение функции.

```
let mult x y z = x+y+z.
```

Определение ЧПФ:

```
let multf = mult 4 5.
```

Применение ЧПФ:

```
let y = multf 2.
```

Ответ – 11.

3.4. Операторы

В функциональных языках используется префиксная и инфиксная форма записи применения (вызова) функций. В *префиксной записи* имя функции располагается перед аргументами. *Инфиксная запись* – это запись символа операции или функции между аргументами. В функциональных языках, как правило, используется префиксная запись. Однако для бинарных операций инфиксная запись является более привычной (удобнее писать `5*6` вместо `mult 5 6`). В связи с этим языки функционального программирования содержат стандартные (встроенные) *операторы* для выполнения бинарных операций; в этом случае операция записывается между своими аргументами.

Кроме использования стандартных операторов, функциональные языки предоставляют возможность определять собственные операторы – функции, предназначенные для инфиксной и префиксной формы записи. В этом случае программы становятся более наглядными.

Для именования операторов используются неалфавитные символы. Например, в языке F# имя оператора – комбинация из следующих литер: ! % & * + - . / < = > & @ ~ | ^.

Пример 3.8. F#. Использование встроенных операторов. Определение собственного оператора возведения числа `x` в степень `n`:

```
let rec (/!) x n =
   if n=1 then x
      else x * ( (/!) x (n-1)).
```

Здесь определяется собственный оператор-функция, обозначенный двумя литерами `/!`, а также используются встроенные операторы сравнения (=) и вычитания (-). В дальнейшем оператор-функция `/!` может быть использован как в инфиксной, так и в префиксной форме записи, например, `2/!3`, `(/!)2 3`.

4. Композиции функций

Как программировать при помощи функций? Ответ на этот вопрос состоит в следующем:

1) определить базовый (основной) набор функций;
2) определить новые функции в терминах исходных.

Построение новых функций из ранее определенных называется *композицией функций* (композиция от лат. *compositio* – составление, соединение частей в единое целое в определенном порядке). Чтобы получить композицию функций, нужно результат одной функции использовать в качестве аргумента другой функции. Таким образом, для составления функциональных программ необходим исходный набор функций и возможность составлять из них композиции. Кроме того, важно помнить, что решение большой задачи не следует оформлять в виде одной большой функции. Решение нужно разбить на несколько функций, а затем, используя композицию, объединить их для получения окончательного результата.

Пусть даны три базовые числовые функции `f(x)`, `g(x)` и `v(x)`. Для получения их композиции `Compos(x) =f(g(v(x)))` надо подставить вместо аргумента функции `g(x)` функцию `v(x)`, а вместо аргумента функции `f(x)- g(v(x))`. В любой композиции первой заканчивает исполнение самая внутренняя функция; здесь – это `v(x)`, затем завершается `g(v(x))`, и, наконец, `f(g(v(x)))`. Таким образом, аргументы композиции функций вычисляются справа налево.

Пример 4.1. F#. Композиция функций нескольких аргументов.

Дана базовая функция `max2` – нахождение наибольшего из двух чисел:

```
let max2 x y =
   if x>=y then x else y.
```

Композиция функций для определения наибольшего из трех чисел:

```
let max3 x y z = max2(max2 x y)z.
```

Функция определения наибольшего из шести чисел `max6` - композиция функций `max2` и `max3`:

```
let max6 x y z a b c =
    max2(max3 x y z)(max3 a b c).
```

Функции могут вкладываться одна в другую на произвольную глубину и, следовательно, составлять произвольно глубокие

композиции. Использование скобок для записи композиций не всегда удобно. В случае большой глубины вложенности функций сложно контролировать соответствие открывающихся и закрывающихся скобок, поэтому программы становятся трудночитаемыми. Современные языки функционального программирования (Haskell, F# и др.) содержат средства для более наглядной записи композиций – *операторы композиции*. В языке F# используются два вида операторов композиции – прямой и обратный. В *прямом операторе композиции* порядок записи функций соответствует порядку их завершения, т.е. функции вычисляются слева направо. В этом случае композиция `f(g(v(x)))` запишется следующим образом: `(v>>g>>f)x`. Если используется *обратный оператор композиции* `(f<<g<<v)x`, то функции вычисляются справа налево.

Оператор композиции принимает на вход два аргумента и выдает результат. Тогда возникает вопрос: что является аргументами и результатом этого оператора? Ответ таков: оператор композиции принимает в качестве аргументов *две функции* и возвращает результат – *λ-функцию*. В теле этой λ-функции реализуется вызов композиции функций-аргументов, записанных в обычной скобочной записи. Оператор композиции – *стандартный* оператор, который определен как обычная функция, используемая для инфиксной записи (см. раздел 3.4):

Определение оператора прямой композиции:

```
let(>>)v g = fun x-> g(v x).
```

Определение оператора обратной композиции:

```
let(<<)g v = fun x-> g(v x).
```

Таким образом, запись `(fun x->g(v x)) 1` эквивалентна записи `(v>>g) 1`.

Кроме того, можно определить и в дальнейшем использовать свой собственный оператор композиции функций. Опишем собственные операторы композиции, которые эквивалентны стандартным операторам >> и <<. Для этого введем обозначения >+ и <+.

Определение оператора прямой композиции:

```
let(>+)v g = fun x-> g(v x).
```

Определение оператора обратной композиции:

```
let(<+)g v = fun x-> g(v x).
```

Теперь имеем по два обозначения для прямого и обратного оператора композиции.

Оператор композиции для трех функций реализуется следующим образом:

```
v>>g>>f =
fun y -> f((fun x -> g (v x)) y).
```

Если для композиции функций задать имя, то получим новую функцию, которую в дальнейшем удобно использовать с различными входными данными.

Пример 4.2. F#. Создание новой функции – результата композиции функций.

Исходные функции:

```
let v x = x+1
let g y = y*2
let f z = z+3.
```

Создание и применение функции `newfunc` – результата композиции трех функций:

```
let newfunc = v>>g>>f
let myresult1 = newfunc 2.
```

Ответ - 9

```
let myresult2 = newfunc 3.
```

Ответ - 11

В рассмотренных примерах бесскобочная запись композиции применялась для функций с одним аргументом. Для бесскобочной записи композиции функций от нескольких аргументов нужно использовать каррирование.

5. Неизменность данных

Функциональная парадигма основана на применении чистых функций, которые являются функциями в математическом смысле. Кроме того, *переменные в функциональном программировании подобны математическим переменным*: они используются только для того, чтобы дать имена значениям аргументов функций и остаются постоянными на протяжении всего вычисления результата.

Хотя в функциональном программировании существует термин "переменная", он понимается здесь иначе, по сравнению с императивными языками. Имена, которые объявляются в функциональной программе, являются *неизменяемыми*. Если однажды переменная получила значение, то в дальнейшем она имеет доступ только для чтения.

Императивное программирование имеет дело с изменяемыми переменными, а функциональное – с неизменяемыми переменными. Тем, у кого есть опыт императивного программирования, отсутствие переменных в традиционном понимании может показаться неудобным.

В функциональном программировании проблема изменения значений переменных решается путем создания новых переменных. Это, в свою очередь, приводит к выделению памяти для множества различных переменных. Поскольку объем памяти ограничен, важно удалять переменные, которые не используются в программе. Освобождение памяти от неиспользуемых переменных выполняет специальная программа *Сборщик мусора*, которая периодически запускается без вмешательства программиста.

Так как функциональная программа использует неизменяемые переменные, она *не имеет состояний*. В функциональных языках *отсутствует присваивание*, т.е. запись `x=x+1` недопустима. Присваивание – понятие императивного программирования, в котором используются переменные с изменяемыми значениями. При объявлении переменных в функциональных языках обычно используется символ =.

Например, именование значения в F# записывается так: `let x=5`. Здесь символ = – это знак равенства в математическом смысле; в императивных же языках, например, C и C# символ = используется для обозначения присваивания.

В функциональных языках *отсутствуют циклы*, так как их использование предполагает изменение значений одной или нескольких переменных (параметров). Повторяющиеся вычисления реализуются через рекурсию. *Рекурсия* – основное средство функционального программирования для организации повторяющихся вычислений.

В отличие от цикла, рекурсия не изменяет значений, она использует новые значения, вычисленные из ранее известных. При реализации повторяющихся вычислений новые значения переменных сохраняются в стеке рекурсивных вызовов. В каждой секции такого стека хранятся копии переменных и/или их новые значения.

6. Типы данных

6.1. Элементарные и составные типы

В функциональном программировании переменные, выражения и функции имеют определенный тип. Обычно используются элементарные и составные типы.

К *элементарным типам* относятся целый, вещественный, символьный, логический, а также пустой тип (обозначает отсутствие результата функции аналогично `void` в C#).

Составные типы – функциональный, строковый, список, кортеж, размеченное объединение. Одна из особенностей функциональных языков – наличие функционального типа.

Функциональный тип – это тип функции. В функциональных языках функции рассматриваются как данные, т.е. функции могут быть аргументами других функций. Кроме того, функции могут создавать и возвращать новые функции. Именно поэтому *функции имеют тип*. Важно понимать, что *тип функции – это описание типов ее аргументов и типа результата,* а не тип значения, которое возвращается функцией. Если аргумент функции `F(x)` имеет тип `N`, а результирующее значение – тип `M`, то говорят, что тип этой функции `N->M`. Обычно эта информация о типе записывается так: `F: N->M`. Типы, в обозначении которых используется символ `->`, являются функциональными типами. *Тип функции нескольких аргументов* удобно представлять с помощью операции декартова произведения. Функция $F(x_1, x_2, \ldots, x_n)$ имеет тип $X_1 \times X_2 \times \ldots X_n \to M$, где X_i – тип аргумента x_i, а `M` – тип результата. Здесь типы аргументов отделены друг от друга символом $\times$. Однако в функциональном программировании используется другой способ записи типа функции. Так как все функции являются каррированными (по умолчанию), т.е. могут быть представлены как последовательность функций одного аргумента, то тип функции записывается так: $X_1 \to (X_2 \to (\ldots\ (X_n \to\ M)\ldots)$. Кроме того, учитывается тот факт, что операция `->` правоассоциативна; поэтому в такой записи скобки не используются.

Таким образом, тип функции нескольких аргументов $F(x_1, x_2, \ldots, x_n)$ в функциональном программировании имеет вид: $X_1 \to X_2 \to \ldots\ X_n \to M$.

Пример 6.1. F#. Тип функции

```
let mult x y = x*y.
```

Тип функции `mult`: `int->int->int` (типы аргументов – `int`, тип результата – `int`).

Список – упорядоченный набор элементов *одного типа*.

Пример 6.2. F#. Определение списка. Тип списка.

Определение списка перечислением всех значений:

```
let x = [1;2;3;4;5].
```

Определение списка с помощью диапазона:
```
let y = [ 1..10 ],
```
`int list` - тип списка из элементов целого типа.

Кортеж – совокупность данных, которая может содержать любое количество элементов любых типов.

Пример 6.3. F#. Определение кортежа.
Тип кортежа
```
let x = (1,2,"aaa","fff",1.2)
```
`int*int*string*string*float` - тип кортежа `x`.

Списки и кортежи используются для представления знаний в задачах искусственного интеллекта, например, информация об игроках теннисного клуба – список кортежей:

```
let club = [("Mary",20,10);
    ("Mikle",22,8);("Ann",18,5)].
```

Расстановка фигур на шахматной доске в программе игры в шахматы – список кортежей, элементами которых являются списки:
```
let cdesk =
   [(["белый";"король"],[4;7]);
    (["черная";"пешка"],[7;8] )].
```

Если функция возвращает несколько значений, то ее результатом является кортеж.

Размеченное объединение определяет множество именованных вариантов, с каждым из которых может быть связан некоторый тип.

Пример 6.4. F#. Использование размеченного объединения для определения перечислимого типа (с вариантами типы не связаны)
```
type Day =
  Monday|Tuesday|Wednesday|
  Thursday|Friday|Saturday|Sunday
```

Рекурсивные размеченные объединения используются для представления деревьев

Пример 6.5. F#. Использование размеченного объединения для представления дерева выражения (с каждым вариантом связан определенный тип).

Узел дерева выражений (рис. 6.1):
✓ элемент, содержащий *операцию* и поддеревья (два поддерева), либо
✓ элемент, содержащий *значение* без поддеревьев – лист.

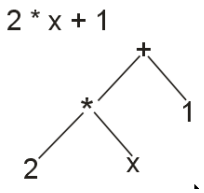

Рис. 6.1. Дерево выражения `2*x+1`

Описание вершины дерева выражения:
```
type ExprTree =
 | Op of char*ExprTree*ExprTree
 | Value of int.
```

Пример 6.6. F#. Использование размеченного объединения для представления дерева поиска (с одним из вариантов связан определенный тип)

Узел дерева поиска (рис. 6.2):
✓ элемент, содержащий значение и два (возможно, пустых) поддерева – левое и правое.
Описание вершины дерева поиска:
```
type BinTree =
     |Node of int*BinTree*BinTree
     |Empty
```

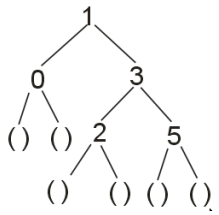

Рис. 6.2. Дерево поиска набора чисел 1,0,3,5,2

6.2. Статическая типизация

Ряд функциональных языков использует статическую проверку типов.
Статическая типизация означает, что
✓ любое значение, выражение, функция имеют строго определенный тип *во время компиляции*,
✓ все проверки соответствия типов выполняется *на этапе компиляции*, а не на этапе выполнения.

Преимущество статической типизации заключается в том, что проверки соответствия типов достаточно выполнить только один раз для всех запусков программы. В случае же динамической типизации такие проверки выполняются при каждом запуске программы. Например, `F#` и `Haskell` – языки со статической типизацией, а `Erlang` использует динамическую проверку типов.

6.3. Строгая типизация

Функциональные языки программирования, как правило, являются строго типизированными. В языках *со строгой типизацией* допускаются только явные преобразования типов (выполняются программистом), а неявные (выполняются компилятором) – запрещены. Это означает, что нельзя применять операцию к операндам различных типов, например, складывать целые и вещественные зна-

чения. В этом случае необходимо всегда делать явное преобразование типов. Для преобразования значений из одного типа в другой используются встроенные функции.

Пример 6.7. F#. Явные преобразования типов
Сложение целой и вещественной констант
`2+int 3.0.`
Тип результата – `int`
`double 2+3.0`
Тип результата – `double.`

6.4. Вывод типов

Большинство функциональных языков поддерживают механизм (концепцию) вывода типов. *Механизм вывода типов* состоит в том, что от программиста не требуется явно указывать типы всех значений, при этом транслятор выводит типы значений из контекста.

При использовании языков со статической типизацией компилятор точно определяет тип каждой конструкции на основе того, как эта конструкция используется в программе.

Пример 6.8. F#. Вывод типов параметров функции. Описание и применение функции
`let isumma a b c = a+b+c.`

Здесь операция `+` может использоваться с операндами разных типов, однако никакой информации о типах параметров нет. Поэтому по умолчанию все параметры и результат функции имеют тип `int`. Следовательно, тип функции `isumma`:

`int->int->int->int.`

Эту функцию можно применять только к значениям целых типов, например, `isumma 1 3 4.` Применение же функции `isumma` к значениям вещественного типа приведет к сообщению об ошибке.

Программист может указать тип значения явно для того, чтобы ограничить использование значения только конкретным типом либо обеспечить более наглядную запись кода.

Пример 6.9. F#. Явное задание типов параметров функции
`let fsumma (a:float) b c = a+b+c.`

Все параметры и результат функции `fsumma` имеют тип `float`. Поэтому тип этой функции – `float->float->float->float` и ее можно применять только к значениям вещественного типа, например:
`let fresult = fsumma 1.2 3.5 2.1.`

Использование механизма вывода типов позволяет уменьшить размер программного кода.

6.5. Параметрический полиморфизм

Парадигма функционального программирования поддерживает *параметрический полиморфизм* – применение одного и того же кода для данных разных типов.

При обработке описания функции компилятор определяет, будет ли эта функция работать с параметрами любых типов. Если это возможно, то *типы параметров* становятся *обобщенными* (универсальными). *Функция,* которая может работать с параметрами любых типов, называется *обобщенной* (универсальной). Использование обобщенных функций позволяет избежать дублирования кода, и, таким образом, сократить размер программы.

Пример 6.10. F#. Описание и применение обобщенной функции.

```
let  max a b  =
    if a>b
      then a else b.
```

Тип функции `max - 'a->'a->'a.` Здесь обобщенный тип обозначен `'a.` Дело в том, что в F# имя типа обобщенного параметра обозначается идентификатором, перед которым стоит апостроф. Обычно в качестве таких идентификаторов используются буквы латинского алфавита. Операцию сравнения можно применять к операндам различных типов, поэтому функция `max` является обобщенной. Ее можно применять с параметрами целого, вещественного, символьного, строкового, логического типа. Однако не все операции являются таковыми.

Ряд операций применяется только к аргументам определенных типов, например, сложение.

Поэтому функции `isumma` и `fsumma` (см. раздел 6.4) не являются обобщенными.

7. Сопоставление с образцом

Языки функционального программирования содержат конструкции для организации ветвлений. Например, в языках Haskell, F# и др. есть конструкция "условное выражение" `if then else` для выбора значения одного из двух возможных выражений. Однако часто приходится выбирать более чем из двух вариантов. В этом случае запись программы с помощью вложенных выражений `if then else` получается достаточно громоздкой. Современные языки функционального программирования содержат более удобную конструкцию для организации вложенных ветв-

лений – *сопоставление с образцом*. Синтаксис этой конструкции неодинаков в разных функциональных языках. В F# сопоставление с образцом состоит из анализируемого выражения и набора правил (вариантов), разделенных символом «|»:

```
match выражение with
| образец_1->выражение_1
| образец_2->выражение_2
. . .
| образец_n->выражение_n.
```

Каждое правило `образец->выражение` определяет один из вариантов вычисления результата; а именно, при совпадении анализируемого выражения с образцом, результатом конструкции является соответствующее выражение. Сравнение значения анализируемого выражения `выражение` с образцами `образец_1,образец_2,...,образец_n` выполняется последовательно сверху вниз до тех пор, пока результат такого сравнения не станет истинным. После этого вычисляется значение соответствующего выражения, которое возвращается в качестве значения всей конструкции. Именно поэтому во всех правилах выражения должны возвращать значения одного типа. Если в процессе сопоставления с образцом не будет найдено совпадение, сгенерируется исключение. Чтобы избежать этого, необходимо определить варианты для всех возможных значений анализируемого выражения.

Пример 7.1. F#. Использование сопоставления с образцом для выбора из нескольких возможных значений

```
match x with
|x when x>0->"число больше 0"
|x when x<0->"число меньше 0"
|_ ->"число равно 0".
```

Здесь символ подчеркивания используется для обозначения произвольного образца.

Конструкцию сопоставления с образцом можно представить, как λ-функцию, аргументом которой является анализируемое выражение:

```
fun arg  ->
   match arg with
   | образец_1->выражение_1
   | образец_2->выражение_2
   . . .
   | образец_n->выражение_n.
```

Результат этой функции – значение выражения одного из правил.

Использование сопоставления с образцом является основным способом обработки сложных структур данных – списков и деревьев.

8. Рекурсия. Хвостовая рекурсия

В функциональном программировании для организации повторяющихся вычислений используются рекурсивные функции, т.е. функции, вызывающие сами себя.

Пример 8.1. F#. Суммирование элементов списка.

Воспользуемся операцией `::` деления списка на голову `(head)` и хвост `(tail)`. Голова – первый элемент списка, а хвост – список без первого элемента.

```
let rec summa l =
    match l with
    |[ ]->0L
    |head::tail->head+summa tail
let res1 = summa [2L;5L;30L;15L].
```

Здесь значение функции формируется при выходе из рекурсивных вызовов. Функция `summa` вызывает сама себя для решения более простых задач, а окончательное решение формируется из возвращаемых результатов. После выхода из текущего рекурсивного вызова в предыдущий вызов нужно выполнить операцию сложения (это есть точка возврата). Поэтому для каждого вызова необходимо сохранять первый операнд операции сложения и точку возврата. Именно с этой целью в процессе рекурсивных вызовов строится стек, в каждой секции которого сохраняются *значения локальных переменных и параметров*, а также *точка возврата.* В дальнейшем они используются для вычисления результата при возврате из рекурсивных вызовов. Стек имеет ограниченный размер. При достаточно большом значении аргумента, и, следовательно, большой глубине вложенности рекурсии возможно переполнение стека, что влечет аварийное завершение работы программы.

Это, в свою очередь, ограничивает объем данных, которые могут обрабатываться рекурсивными функциями. Именно по причине переполнения стека аварийно завершится суммирование элементов списка:

```
let res2 =summa [1L..100000L].
```

Для решения проблемы переполнения стека в функциональном программировании широко используется *хвостовая рекурсия* – особая форма рекурсии, которая не использует стек для организации рекурсивных вызовов. Дело в том, что в этом случае не возника-

ет необходимости сохранять параметры функции, локальные переменные и адрес возврата каждого рекурсивного вызова. Следовательно, глубина вложенности рекурсии может быть произвольной. Функции, реализующие хвостовую рекурсию, имеют особый способ организации. Во-первых, рекурсивный вызов *единственный* и является *последним действием функции*. Во-вторых, в функции используется дополнительный параметр, который называют *аккумулятором (сумматором)*. В процессе рекурсивных вызовов на аккумуляторе постепенно накапливается значение функции. Поэтому хвостовую рекурсию называют также *методом накапливающего параметра*.

Пример 8.2. F#. Суммирование элементов списка с использованием хвостовой рекурсии

```
let rec h_summa x (acc:int64)=
    match x with
    |[]->acc
    |h::t->h_summa t (h+acc)
let z1=h_summa [1L..100000L] 0L.
```

Чтобы вычислить сумму элементов списка, нужно вызвать функцию с нулевым значением аккумулятора. Результат функции постепенно накапливается на аккумуляторе при увеличении уровня вложенности рекурсии. В функции `h_summa` рекурсивный вызов – последнее действие функции. Суммирование значения первого элемента списка и аккумулятора выполняется до рекурсивного вызова. Окончательный результат функции получается на аккумуляторе рекурсивного вызова с наибольшей глубиной вложенности. Выход из рекурсивного вызова глубины вложенности `i(i=0,n)` осуществляется без выполнения каких-либо дополнительных действий с результатом рекурсивного вызова глубины `i+1` (сравните с функцией без хвостовой рекурсии, в которой выполнялась операция сложения после возврата из очередного рекурсивного вызова). Отсюда следует, что *не нужно запоминать точку возврата, значения локальных переменных и параметров*, так как теперь они не используются для вычисления значения функции при возврате из очередного рекурсивного вызова. Следовательно, стек не нужен. Теперь в результате суммирования элементов списка `[1L..100000L]` получим число `5000050000L`. Использование же обычной рекурсии для решения этой задачи приводит к переполнению стека (пример 8.1).

Программисту, использующему функцию с хвостовой рекурсией, нет необходимости знать о наличии аккумулятора. Поэтому, как правило, создается объемлющая функция, которая содержит описание и вызов функции с аккумулятором.

Пример 8.3. F#. Суммирование элементов списка. Использование объемлющей функции.

```
let summa x =
   let rec h_summa x (acc:int64)=
       match x with
       |[]->acc
       |h::t->h_summa t (h+acc)
   h_summa x 0L
let z = summa [1L..100000L].
```

Теперь применим хвостовую рекурсию в случае, когда функция возвращает несколько результатов.

Пример 8.4. F#. Результат рекурсивной функции – сумма и произведение списка чисел.

В этом случае используется два аккумулятора, а функция возвращает кортеж:

```
let sum_pr x =
    let rec help x acc_s acc_pr =
        match x with
        |[] -> (acc_s,acc_pr)
        |h::t -> help t (acc_s+h)
                        (acc_pr*h)
    help x 0 1
let res = sum_pr [2;5;30;10].
```

Ответ: `(47,3000)`.

Преимущества функции с хвостовой рекурсии:

✓функция выполняется быстрее, так как отсутствуют операции со стеком;

✓количество рекурсивных вызовов не ограничено, так как стек не используется.

Обычно трансляторы функциональных языков распознают хвостовую рекурсию и реализуют ее без использования стека. Возникает вопрос: любую ли рекурсивную функцию можно преобразовать в функцию с хвостовой рекурсией? Ответ на этот вопрос отрицательный. Дело в том, что метод накапливающего параметра не является универсальным. Это – специальный вид рекурсии, в котором используется единственный рекурсивный вызов, являющийся последним действием функции. Если функция содержит два рекурсивных вызова, один из них не удастся сделать хвостовым.

9. Функции высших порядков

В функциональных языках функции рассматриваются как данные.

Функция высшего порядка – функция, которая использует функции в качестве аргументов и/или возвращает некоторую функцию в качестве результата.

Всякая функция с более чем одним аргументом *при каррировании* становится функцией высшего порядка (ФВП), так как является функцией первого аргумента и возвращает функцию от оставшихся аргументов (см. раздел 3.3). *Оператор композиции функций* также является ФВП, так как он использует в качестве аргументов две функции и возвращает результат – λ-функцию (см. раздел 4).

Рассмотрим несколько примеров функций высших порядков (ФВП).

Пример 9.1. F#. Обобщение нескольких функций.

Пусть заданы три функции для вычисления суммы, суммы квадратов, а также суммы кубов чисел в диапазоне от `a` до `b`.

```
//сумма чисел от a до b
let rec sum a b =
    if a>b then 0
      else a + sum (a+1) b
//сумма квадратов чисел от a до b
let rec sumsq a b =
    if a>b then 0
      else a*a + sumsq (a+1) b
//сумма кубов чисел от a до b
let rec sumcub a b =
    if a>b then 0.
      else a*a*a + sumcub (a+1) b
```

Эти функции отличаются только первым слагаемым выражения ветви `else`. Заменим три функции одной ФВП:

```
let rec sumCommon f a b =
   if a>b then 0
    else f a+sumCommon f (a+1) b.
```

Первый аргумент функции `sumCommon` – функция для вычисления первого слагаемого выражения ветви `else` исходных функций, а второй и третий аргументы задают наименьшее и наибольшее число диапазона.

Теперь применим ФВП `sumCommon`:

```
let result1 =
    sumCommon (function x->x)2 4.
```

Ответ – `9.`

```
let result2 =
    sumCommon(function x->x*x)2 4
```

Ответ – `29.`

```
let result3 =
    sumCommon(function x->x*x)3 5
```

Ответ – `50.`

```
let result4 =
  sumCommon(function x->x*x*x)2 4
```

Ответ – `99.`

Использование ФВП позволяет избавиться от повторяющихся фрагментов нескольких функций. Код становится более коротким и удобным в использовании, так как отлаживать и тестировать теперь придется одну функцию вместо трех.

Пример 9.2. F#. Вычисление производной.

Опишем функцию вычисления производной:

```
let deriv(f:float -> float) x =
    let dx= 0.001;
    (f (x+dx) - f(x-dx))/(2.0*dx)
```

Аргументом ФВП `deriv` является *функция*, производную которой нужно вычислить. Воспользуемся функцией `deriv`. Для этого определим функцию `q(x)` и вычислим значение ее производной в точке `2.0`:

```
let q x = x**3.0-x-1.0
deriv q 2.0.
```

Ответ – `11.0.`

Теперь найдем значение производной функции `f(x)` = x^3 в точке `7.0`:

```
deriv (fun x -> x*x*x) 7.0.
```

Ответ – `147.0.`

Таким образом, используя ФВП `deriv` с аргументом-функцией, получаем возможность вычислять значение производной для различных функций.

Зафиксируем функцию `f(x)` = x^2, для которой будет вычисляться значение производной в разных точках. Для этого выполним каррирование функции `deriv`:

```
let  deriv1 = deriv(fun x -> x*x)
```

и найдем значение производной функции `f(x)` = x^2 в разных точках:

```
deriv1 3.0
deriv1 9.0
deriv1 81.0.
```

Каррирование ФВП `deriv` позволяет сократить объем кода при вычислении производной какой-либо функции в различных точках.

Пример 9.3. F#. Вычисление вложенных функций `f(f(f(f(x)…)`.

Рассмотрим описание функции `nest`, которая `n` раз применяет функциональный аргумент `f` к заданному значению `x` (используем хвостовую рекурсию):

```
let rec nest n f x =
    match n with
     |0 -> x
     |n -> nest(n-1) f (f x).
```

Трижды применим функцию `f(x)=3*x+1` к значению 2, т.е. `3*(3*(3*2+1)+1)+1`:

```
let v = nest 3 (fun x -> 3*x+1) 2
```

Ответ – *67*.

Пример 9.4. F#. Вычисление второй и третьей производной функции.

Воспользуемся вложенной функцией `deriv` (см. примеры 9.2 и 9.3) для вычисления второй и третьей производной функции $g(x)=3x^3+2x^2+4$ в точке `2.0`.

```
let g (x:float)=
            3.0*x*x*x+2.0*x*x+4.0
```

Вычисление `g'(g'(2.0))`:

```
let y=nest 2 deriv g 2.0.
```

Ответ – `40.0`.

Вычисление `g'(g'(g'(2.0)))`:

```
let y1=nest 3 deriv g 2.0.
```

Ответ – `18.0`.

Здесь аргумент ФВП `nest` – ФВП `deriv`, аргументом которой является функция.

ФВП для работы со списками

В функциональных языках при работе со *списками* достаточно часто используются операции *отображения*, *фильтрации* и *свертки*. Функции, реализующие эти операции, принимают в качестве аргументов другие функции, т.е. являются ФВП.

Функция отображения `map` применяет функцию-аргумент к каждому элементу списка и, таким образом, создает новый список.

Пример 9.5. F#. Использование функции отображения `map`.Создание нового списка возведением в куб элементов исходного списка.

Модуль `List` стандартной библиотеки F# содержит множество функций для реализации операций со списками.

```
let v = List.map
          (fun x -> x*x*x) [1..5]
```

Ответ - `[1;8;27;64;125]`.

Здесь аргумент `map` – анонимная функция.

Функция `filter` создает список, содержащий только те элементы исходного списка, для которых функция-аргумент возвращает значение "истина".

Пример 9.6. F#. Использование функции фильтрации `filter`. Создание нового списка из нечетных элементов исходного списка.

```
let x2=List.filter
           (fun x->x%2<>0)[1..10]
```

Ответ - `[1;3;5;7;9]`.

Операция свертки применяется в тех случаях, когда в результате обработки исходного списка необходимо получить единственное число (максимальный или минимальный элемент, сумму или произведение элементов и т.д). Функция `List.folder` принимает функцию `f` и список $[x_1, x_2, \ldots x_n]$, а вычисляет `f (…f(f(`s_0`,`x_1`),`x_2`),…`x_n`)`, где s_0 – начальное значение аккумулятора в соответствующем итерационном алгоритме.

Пример 9.7. F#. Использование функции `folder`. Вычисление суммы элементов списка

```
let summa=List.fold(fun s x->s+x)
                    0 [1;2;3;4;5]
```

Ответ – `15`.

10. Ленивые вычисления

В императивных языках значение выражения вычисляется независимо от того, будет ли оно использоваться в дальнейшем. Поэтому во время вызова функции вычисляются все ее аргументы. Если какой-либо аргумент не будет использован в теле функции, это означает, что его вычисление выполнено напрасно. Такая стратегия вычислений называется *энергичной*. Кроме того, существует *ленивая* стратегия, в соответствии с которой вычисление выражения откладывается до тех пор, пока не потребуется его значение. В этом случае неиспользуемые значения не вычисляются. Энергичная и ленивая стратегии – важнейшие понятия функционального программирования. Языки, использующие ленивую стратегию, называются *нестрогими* (*ленивыми*).

Например, язык Haskell – нестрогий; язык же F# – строгий, так как поддерживает энергичную стратегию. Кроме того, в F# имеются средства для реализации ленивой стратегии, т.е. программист сам принимает решение, какие вычисления выполнять лениво, а какие – энергично. В этом случае для откладывания вычислений используется ключевое слово lazy, а для получения вычисленного значения – вызов метода Force. Если `выражение` имеет тип `T`, то конструкция `lazy(выражение)` возвращает значение типа `Lazy<T>`.

Пример 10.1. F#. Использование ленивых вычислений.

```
open System
let z=lazy
        int (Console.ReadLine())
let x = lazy (10*z.Force())
let y = lazy (40+z.Force())
let c1 =
  if z.Force() > 25
     then x.Force()*x.Force()
         else y.Force()*y.Force()
```

Значение каждого выражения вычисляется только один раз в результате первого вызова метода `Force` и сохраняется для дальнейшего использования. Именно поэтому только один раз вычисляется значение одной из переменных переменной `x` или `y`; только один раз запрашивается значение переменной `z`. Несмотря на то, что ленивые вычисления позволяют сократить общий объем вычислений, нередко их постоянное использование затрудняет понимание программ.

Заключение

В статье рассмотрены различия императивного и функционального стилей программирования; систематизированы принципы функционального программирования: концепция λ-исчисления, особенности чистых функций, каррирование, композиции функций, неизменность данных, типы данных, вывод типов, параметрический полиморфизм, сопоставление с образцом, хвостовая рекурсия, функции высших порядков, а также ленивые вычисления.

Список литературы

1. *Бердж В.* Методы рекурсивного программирования. М.: Машиностроение, 1983. 248 с.
2. *Хендерсон П.Ф.* Функциональное программирование. Применение и реализация. М.: Мир, 1983. 605 с.
3. *Хьювенен Э., Сеппанен Й.* Мир Лиспа. М.: Наука, 1994. Т. 1, 2. 458 с. (Т. 1), 332 с. (Т. 2).
4. *Джон Харрисон.* Введение в функциональное программирование. Кембридж, 1997. 161 с.
5.. *Филд А., Харрисон П.* Функциональное программирование. М.: Мир, 1993. 638 с.
6. *Себеста Р.У.* Основные концепции языков программирования. М.; СПб.; Киев: Изд. Дом Вильямс, 2001. 672 с.
7. *Городняя Л.В.* Основы функционального программирования. М.: Интернет-университет информационных технологий, 2004.
8. Robert Pickering. Foundations of F#. Apress, 2007. 345с.
9. *John Harrop.* F# for Scientists. A John Wiley & Sons, INC., 2008. 334с.
10. *Jon Skeet, Tomas Petricek*. RealWorld Functional Programming with Examples in F# and C#. N-Y.: Manning Publications, 2009. 501 с.
11. *Кирпичев Е.* Элементы функциональных языков. Практика функционального программирования. 2009. Вып. 3. С. 114–274.
12. *Кубенский А.А*. Функциональное программирование. СПб.: СПбГУ ИТМО, 2010. 251 с.
13. *D. Syme, A. Granicz, A.Cisternio*. Expert F# 2.0. Apress, 2010. 624 с.
14. *Крис Маринос*. Введение в функциональное программирование для .NET-разработчиков. MSDN Magazine, 2010. 20 с.
15. *Лазин Е., Моисеев М., Сорокин Д.* Введение в F# // Практика функционального программирования. Т. 5, вып. 5. 2010. 39–78 с.
16. *Д. Сошников.* Функциональное программирование на F#. М.: ДМК, 2011. 192 с.
17. *Душкин Р.В*. Функциональное программирование на языке Haskell. М.: ДМК Пресс, 2016. 608 с.
18. *Зыков С.В.* Введение в теорию программирования: учеб. пособие. М.: ИНТУИТ, 2018. 396 с.
19. *Рубно-Санчес Мануэль*. Введение в рекурсивное программирование. М.: ДМК Пресс, 2019. 438 с.
20. F# *Basic Tutorial*. https://www.tutorialspoint.com/fsharp/index.htm. (дата обращения: 22.03.2020).

Principles of functional programming

L. A. Zalogova

Perm State University; 15, Bukireva st., Perm, 614990, Russia
zalogova.la@gmail.com

Functional programming is currently experiencing an extensive development. In comparison with the imperative programming, the functional approach is more efficient in solving some types of tasks. The paper considers and systematizes the principles which are typical for different functional languages. The paper could be of interest to those who are skilful in imperative programming and have an intention to know more

about the functional languages. Pascal procedural language and F# functional language are used for illustrative purposes.

Keywords*: pure functions; function composition; data immutability; higher order functions; tail recursion.*