9. Функциональное программирование

Математические функции выражают связь между параметрами (входом) и результатом (выходом) некоторого процесса. Так как вычисление – это тоже процесс, имеющий вход и выход, функция является вполне подходящим и адекватным средством описания вычислений. Именно этот простой принцип положен в основу функционального программирования.

Функциональное программирование – это подход к программированию, при котором программа задаётся совокупностью определений функций без явного указания последовательности их применения.

Понятие *функция* связано с понятиями:

- аргумента функции,
- области её существования и значения,
- соответствия между её аргументами и результатами,
- применения функции к её аргументам.

Единственным действием в программе является вызов функции. Единственным способом разделения программы на части – введение имени для функции и задание для этого имени выражения, которое вычисляет значение функции. В процессе выполнения программы функции получают параметры, вычисляют и возвращают результат, в случае необходимости вычисляя значения других функций. Чтобы программировать при помощи функций, нужно иметь возможность определить достаточно богатое множество базовых функций и затем использовать композицию, чтобы определять новые функции в терминах исходных.

Функциональное программирование не использует концепцию памяти как хранилища значений переменных. Операторы присваивания отсутствуют, вследствие чего переменные обозначают не области памяти, а объекты программы, что полностью соответствует понятию переменной в математике. Кроме того, нет существенных различий между программами и данными. В результате этого функция может быть значением вызова другой функции или элементом структурированного объекта.

Есть две основные абстракции функционального программирования:

1. *Отказ от управляющих операторов*. Вместо них ко всей программе единообразно применяется механизм управления. Программа рассматривается как набор функций, а вычисление – как попытка вычислить целевую функцию на основе известных программе функций.
2. *Не используются операторы присваивания и явные указатели*. Вместо этого для декомпозиции структур данных используется обобщённый механизм сопоставления с образцом. Объекты нельзя изменять и уничтожать, можно только создавать новые путём декомпозиции и синтеза существующих объектов.

Функциональные языки обладают двумя весьма мощными средствами:

1. *Способность структурировать данные*. Мощность этого средства состоит в том, что структуры данных выступают как единые значения. Они могут вводиться в функцию как аргументы и выдаваться из неё как результаты. Наиболее важно, что однажды построенная структура **не может изменяться.**
2. *Возможность определять функции высших порядков*. Такие функции имеют другие функции либо своими аргументами, либо вырабатывают в качестве результатов. Использование функций высших порядков приводит к более кратким и сжатым программам.

9.1. Математические основы функционального программирования

К фундаментальным математическим исследованиям, которые привели к появлению функционального подхода к программированию, относятся:

1. *Моделирование функций*. Разработанная *А. Чёрчем* теория конечных последовательностей в форме λ-конверсий положила начало математическому исчислению, формализующему понятие функции. Выбирая λ-исчисление как теоретическую модель, *Д. Мак-Карти* предложил рассматривать функции как общее базовое понятие, к которому достаточно естественно могут быть сведены все другие понятия, возникающие при программировании. Управление обработкой информации в λ-исчислении осуществляется в рамках иерархии свободных и связанных переменных, реализуемых с помощью таблицы соответствия символов. Обработка представляется посредством интерпретации выражений, построенных из определённых функций.
2. *Моделирование типов*. Посвящённые типизированному λ-исчислению работы, согласно которым аргументам функций и самим функциям можно назначать (приписывать) тот или иной тип. Типизация существенно увеличивает вычислительную стройность и значимость любой математической формализации, и естественно, без неё немыслимы современные языки программирования. Функциональное программирование поддерживает универсальные методы обработки разнотипных объектов.
3. *Моделирование среды вычислений в форме абстрактной машины*. Абстрактной машиной *(abstract machine)* называют математическую формализацию, которая моделирует правила выполнения программы (иначе алгоритмы) для того или иного реального компьютера. В настоящее время при практической реализации функциональных и объектно-ориентированных языков широко используются аналоги абстрактных машин в форме так называемых *виртуальных* машин. Виртуальные машины представляют собой средство создания про-

межуточного (следующего за текстом программы на высокоуровневом языке) кода (например, Java-кода), который затем транслируется в машинный код. Абстрактные машины позволяют адекватно моделировать различные подходы и стратегии вычислений (включая рекурсивные), а также вычисления по необходимости. *Х. Карри* предложил теорию функций без свободных переменных (иначе называемых *комбинаторами*), известную как комбинаторная логика. Эта теория представляет собой формальный язык, аналогичный языку функционального программирования и позволяющий моделировать вычисления в среде абстрактных машин.

4. *Моделирование вычисления значений*. Теория решёток *Д. Скотта* стала основой для моделирования вычисления значения функции. Он предложил денотационный подход к семантике, согласно которому для построения теории вычислений сначала необходимо перечислить стандартные (или наиболее часто используемые в рамках конкретной формализации) домены. После перечисления стандартных доменов необходимо определить конечные домены (элементы конечных доменов можно перечислить явным образом). Наконец, после перечисления доменов определить конструкторы доменов, под которыми понимаются операции построения новых доменов на основе имеющихся. Денотационный подход предполагает анализ синтаксически корректных конструкций языка (денотатов) с точки зрения вычисления их значений посредством специализированных функций.

9.2. Свойства функциональных языков

К основным свойствам функциональных языков относятся следующие:

- краткость и простота;
- строгая типизация;
- модульность;
- функции – это значения;
- чистота (отсутствие побочных эффектов);
- отложенные (ленивые) вычисления.

9.2.1. Краткость и простота

Программы на функциональных языках обычно намного короче и проще, чем те же самые программы на традиционных языках программирования. *Пример*. Программа быстрой сортировки Хоара на языке C.

```
void quickSort (int a[], int l, int r)
{
      int i = l ;    int j = r ;    int x = a[(l + r) / 2] ;
      do {
            while (a[i] < x) i++ ;
            while (x < a[j]) j-- ;
```

```
            if (i <= j) {
                int temp = a[i] ;
                a[i++] = a[j] ;
                a[j--] = temp ;
            }
        } while (i <= j) ;
        if (l < j) quickSort (a, l, j) ;
        if (i < r) quickSort (a, i, r) ;
    }
```

Пример. Программа быстрой сортировки Хоара на абстрактном функциональном языке.

```
quickSort ([ ]) = [ ]
quickSort ([h : t]) =
    quickSort (n | n ⊆ t, n <= h) + [h] + quickSort (n | n ⊆ t, n > h)
```

Текст программы следует читать так:

1. Если список пуст, то результатом также будет пустой список.
2. Если список не пуст, выделяется голова (первый элемент) и хвост (список из оставшихся элементов). В этом случае результатом будет являться конкатенация отсортированного списка из всех элементов хвоста, которые меньше либо равны голове, списка из самой головы и списка из всех элементов хвоста, которые больше головы.

Пример. Программа быстрой сортировки Хоара на языке Haskell.

```
quickSort [ ] = [ ]
quickSort (h : t) = quickSort [y | y <- t, y < h] ++ [h]
  ++ quickSort [y | y <- t, y >= h]
```

Как видно, даже на таком простом примере функциональный стиль программирования выигрывает и по количеству написанного кода и по его элегантности.

Полезным свойством, позволяющим сократить программу, является встроенный механизм сопоставления с образцом. Это позволяет описывать функции как индуктивные определения. Например, программа определения **N**-го числа Фибоначчи:

```
fibb (1) = 1
fibb (2) = 1
fibb (N) = fibb (N – 2) + fibb (N – 1)
```

9.2.2. Строгая типизация

Практически все современные языки программирования являются строго типизированными языками (возможно, за исключением JavaScript и его диалектов). Строгая типизация обеспечивает безопасность. Программа, прошедшая проверку типов, просто не может завершиться с сообщением *"access violation"*. В функциональных языках бОльшая часть ошибок мо-

жет быть исправлена на стадии компиляции, поэтому стадия отладки и общее время разработки программ сокращаются. Кроме того, строгая типизация позволяет компилятору генерировать более эффективный код, и тем самым ускорять выполнение программы.

Функциональные языки поддерживают параметрический полиморфизм. В примере быстрой сортировки Хоара на C функция сортирует список значений типа целых чисел *(int)*, а функция на абстрактном функциональном языке – список значений любого типа, который принадлежит к классу упорядоченных величин. Поэтому последняя функция может сортировать и список целых чисел, и список чисел с плавающей точкой, и список строк. Можно описать какой-нибудь новый тип. Определив для этого типа операции сравнения, возможно без перекомпиляции использовать функцию **quickSort** так же и со списками значений этого нового типа.

Ещё одной разновидностью полиморфизма является перегрузка функций, позволяющая давать различным, но в чём-то схожим функциям одинаковые имена. Некоторые функциональные языки помимо параметрического полиморфизма, поддерживают и перегрузку функций.

В традиционных объектно-ориентированных языках имеются шаблоны, которые позволяют определять полиморфные функции. Но шаблоны C++ на самом деле порождают множество перегруженных функций, которые компилятор должен каждый раз компилировать, что неблагоприятно сказывается на времени компиляции и размере кода. В функциональных языках полиморфная функция – это одна единственная функция.

В некоторых языках строгая типизация требует явного описания типов всех значений и функций. Чтобы избежать этого, в строго типизированные функциональные языки встроен специальный механизм, позволяющий компилятору определять типы констант, выражений и функций из контекста. Этот механизм называется механизмом вывода типов. В большинстве случаев можно не указывать типы функций.

9.2.3. Модульность

Механизм модульности позволяет разделять программы на несколько сравнительно независимых частей (модулей) с чётко определёнными связями между ними. Тем самым облегчается процесс проектирования и последующей поддержки больших программных систем. Поддержка модульности не является свойством именно функциональных языков программирования, однако поддерживается большинством таких языков.

9.2.4. Функции – это значения

В функциональных языках, как и в математике, функции могут быть переданы другим функциям в качестве аргумента или возвращены в качестве результата.

Если аргумент используется в функции как объект, участвующий в вычислениях, то это данные.

Если аргумент используется как средство, определяющее вычисления, то это функция.

Функции, принимающие функциональные аргументы, называются функциями высших порядков или функционалами.

Самый известный функционал – функция **map**. Эта функция применяет некоторую функцию ко всем элементам списка, формируя из результатов заданной функции другой список. Например, определив функцию возведения целого числа в квадрат:

```
square (N) = N * N
```

можно воспользоваться функцией **map** для возведения в квадрат всех элементов некоторого списка:

```
squareList = map (square, [1, 2, 3, 4])
```

Результатом выполнения этой инструкции будет список **[1, 4, 9, 16]**.

9.2.5. Чистота

В традиционных языках функция в процессе своего выполнения может читать и модифицировать значения глобальных переменных и осуществлять операции ввода/вывода. Поэтому, если вызвать одну и ту же функцию дважды с одним и тем же аргументом, может случиться так, что в качестве результата будут получены два различных значения. Такая функция называется функцией с побочными эффектами.

Описывать функции без побочных эффектов позволяет практически любой язык. Однако некоторые языки поощряют функции с побочными эффектами. Например, во многих объектно-ориентированных языках методы класса используют скрытый параметр **this**, который эти методы могут неявно модифицировать.

В функциональном программировании оператор присваивания отсутствует, объекты нельзя изменять и уничтожать, можно только создавать новые путём декомпозиции и синтеза существующих объектов. Благодаря этому в функциональных языках все функции свободны от побочных эффектов.

9.2.6. Отложенные вычисления

В традиционных языках программирования вызов функции приводит к вычислению всех аргументов. Этот способ вызова функции называется *вызовом по значению*. Если какой-либо аргумент не использовался в функции, то результат вычислений пропадает, следовательно, вычисления были произведены впустую. Противоположностью вызову по значению является *вызов по необходимости*, который реализует отложенные вычисления. В

этом случае аргумент вычисляется, только если он необходим для получения результата. Примером из C++ является оператор конъюнкции (&&), который не вычисляет значение второго аргумента, если первый имеет ложное значение.

Если функциональный язык не поддерживает отложенные вычисления, то он называется строгим. Языки, использующие отложенные вычисления, называются нестрогими. Некоторые функциональные языки поддерживают дополнительное зарезервированное слово **lazy** и конструкцию для списков значений, вычисляемых по необходимости.

9.3. Преимущества и недостатки функционального подхода

Основные преимущества функционального подхода к программированию:

1. *Полностью автоматическое управление памятью компьютера*. Реализации функциональных языков поддерживают автоматизированное динамическое распределение памяти компьютера для хранения данных. При этом программист избавляется от рутинной необходимости контролировать данные, а при необходимости может запустить функцию сборки мусора – очистки памяти от тех данных, которые больше не потребуются программе.
2. *Простота повторного использования фрагментов кода*. Сложные программы при функциональном подходе строятся посредством агрегирования функций. При этом текст программы представляет собой функцию, некоторые аргументы которой можно так же рассматривать как функции. Таким образом, повторное использование кода сводится к вызову ранее описанной функции, структура которой прозрачна математически.
3. *Расширенная поддержка параметрических функций и абстрагирование* от машинного представления данных. Типы отдельных переменных функций, используемых в функциональных языках, могут быть переменными. При таком подходе обеспечивается возможность обработки разнородных данных, или полиморфизм. Таким образом, при создании программ на функциональных языках программист сосредотачивается на предметной области и в меньшей степени заботится о рутинных операциях (например, обеспечении правильного с точки зрения компьютера представления данных).
4. *Прозрачность реализации рекурсивных функций*. Поскольку функция является естественным формализмом для функциональных языков, реализация различных связанных с функциями аспектов программирования существенно упрощается. Интуитивно прозрачным становится написание рекурсивных функций (вызывающих самих себя в качестве аргумента). Естественной становится и реализация обработки рекурсивных структур данных (таких как списки и деревья).

5. *Символьная обработка*. Благодаря реализации механизма сопоставления с образцом функциональные языки весьма хорошо применимы для символьной обработки.
6. *Параллелизм*. Поскольку все функции для вычислений используют только свои параметры, можно вычислять независимые функции в произвольном порядке или параллельно, на результат вычислений это не повлияет.

К недостаткам функционального подхода относят следующее:

1. *Необходимость фундаментальной математической подготовки разработчиков*. Для профессиональной разработки программного обеспечения на функциональных языках необходимо глубоко понимать природу функции.
2. *Нелинейная структура программы*. Недостаток достаточно субъективен.
3. *Отсутствие стандартов*. До настоящего времени не существует стандартов программирования для функциональных языков. Кроме того, распространено множество диалектов, каждый из которых реализован по-своему, что ограничивает совместимость программ.

9.4. λ-исчисление как формализация языка функционального программирования

λ-исчисление было построено для формализации понятия *функция*, поэтому его можно рассматривать как математическую формализацию, моделирующую языки функционального программирования. Эта теория известна также под другим, более точно характеризующим её суть названием: исчисление λ-конверсий.

λ-исчисление – математическая теория, предназначенная для формализации обозначения и переобозначения (иначе конверсии).

Под *конверсией* понимается преобразование объектов исчисления (а в программировании – *функций* и данных) из одной формы в другую.

Конверсии обеспечивают переход к вновь введённым обозначениям, и таким образом позволяют представлять предметную область в более компактном или более детальном виде (говоря математическим языком, изменять уровень *абстракции* по отношению к предметной области).

Основным объектом λ-исчисления является *функция*, именно поэтому эта теория весьма продуктивна при моделировании языков функционального программирования.

λ-исчисление есть язык для определения функций. Выражения языка называются *λ-выражениями* и каждое такое выражение обозначает функцию.

9.5. Редукция и стратегии редукций

Для *λ-выражений* определена только одна операция – операция *редукции*, то есть упрощения вида.

Согласно аксиомам и правилам вывода можно утверждать, что в любом выражении можно выделить вхождения подвыражения и заменить их все любой редукцией из этого подвыражения.

В функциональном программировании программа представляет собой совокупность описаний функций и выражения, которые необходимо вычислить. Это выражение вычисляется посредством редукции (серии упрощений) до тех пор, пока это возможно по следующим правилам:

- вызовы базовых функций заменяются соответствующими значениями;
- вызовы не базовых функций заменяются их телами, в которых параметры замещены аргументами.

В контексте функциональных языков и *λ-исчисления* существуют две важные стратегии редукций:

1. *Аппликативный порядок редукций* (*Innermost reduction*). На каждом шаге редукции выбирается самый *внутренний редекс*[1] (такой, который не содержит внутри себя других *редексов*). Аппликативная стратегия не всегда позволяет получить нормальную форму выражения и может привести к бесконечному процессу.
2. *Нормальный порядок редукций (Outermost reduction).* На каждом шаге редукции выбирается самый *внешний редекс* (тот, который не содержится внутри других редексов). Доказано, что нормальная стратегия гарантирует получение нормальной формы выражения, если она существует.

Аппликативный порядок редукций можно представлять себе как порядок, при котором в каждом вызове функции, прежде всего, *вычисляется* значение аргумента вызова, а потом уже происходит подстановка этого значения в тело вызываемой функции. Таким образом, этот порядок редукций соответствует *энергичному способу вычисления* значения функций в функциональном программировании. Если вычисление аргумента приводит к аварии или бесконечному циклу, то вычисление выражения приводит к тому же, хотя возможно этот аргумент и не требовался для получения результата.

---

[1] *redex* – сокращение словосочетания *reducible expression*

При нормальном порядке редукций подстановка аргумента в тело функции происходит до того, как преобразования будут производиться над самими аргументами. Такая схема преобразования выражения похожа на процесс *ленивых вычислений* в функциональных языках программирования. В схеме применения редукций к выражениям в λ-исчислении процесс преобразования выражения, которое служит аргументом применения функции, будет происходить столько раз, сколько раз аргумент появляется в определении этой функции. В схеме ленивых вычислений в функциональных языках программирования вычисление аргумента осуществляется лишь однажды (при первом обращении к аргументу), в дальнейшем происходит обращение к уже вычисленному значению. В программировании *нормальная редукционная стратегия* соответствует *вызову по имени*. Это значит, что аргумент выражения не вычисляется до тех пор, пока к нему не возникнет обращения в теле выражения.

В зависимости от стратегии редукций различают:

1. *Жадные вычисления (Eager evaluation)*, которые используют аппликативный порядок редукции. Например, в **f(g(z(x)))** первым будет вычислен **x** (если он является редексом), затем **z(x)** и т. д.
2. *Ленивые вычисления (Lazy evaluation)*, которые используют нормальный порядок редукций, при котором не происходит дублирования вычислений. Например, в **f(g(z(x)))** первым будет редуцирован **f(невычисленныйАрг)**, где **невычисленныйАрг = g(z(x))**.

Вполне разумно вычислять аргументы только тогда, когда они потребуются. В такой же степени разумно вычислять аргумент только один раз, даже если он потребуется больше чем однажды. Это может быть достигнуто оптимизацией. Выражение **f (1+ 2 + 3)** могло бы вычисляться как-нибудь так:

```
f (1 + 2 + 3)
x + x + x {x = 1 + 2 + 3}
x + x + x {x = 3 + 3}
x + x + x {x = 6}
6 + 6 + 6
12 + 6
18
```

Этот приём *вынесения за скобки* повторных вычислений аргументов и называется ленивыми вычислениями. Ленивые вычисления – эффективный метод реализации нормальной стратегии.

9.6. Сравнение функционального и логического программирования

Функциональная и логическая программа представляют собой набор правил подстановки.

Алгоритм приведения к нормальной форме в функциональном программировании:

```
пока выражение содержит редексы
        выбрать редекс
        применить к нему подстановку
        заменить редекс результатом подстановки
(теперь выражение в нормальной форме)
```

Алгоритм для редукции целей в логическом программировании:

```
пока цель не стёрта
        выбрать подцель
        выбрать подходящее правило подстановки
        если такое правило есть
                отметить точку возврата
                применить правило подстановки
                заменить подцель результатом подстановки
        иначе
                если определена точка возврата
                        вернуться к запомненному состоянию
                иначе
                        неудача
        успех
```

Существует очень близкая связь между математикой функций и логикой, потому что

$$y = f(x_1, \ldots, x_n)$$

эквивалентно истинности логической формулы:

$$\forall\, x_1, \ldots, x_n \; \exists\, !\, y \,(y = f(x_1, \ldots, x_n))$$

, где $\exists\, !\, y$ означает: *существует единственное значение* y.

Между функциональным и логическим программированием имеются следующие различия:

1. Логическое программирование использует двунаправленное сопоставление, что сильнее однонаправленного сопоставления с образцом, используемого функциональным программированием.
2. Функциональные программы являются однонаправленными в том смысле, что программа возвращает значение, получив все аргументы. В логических программах любой из аргументов цели может остаться неопределённым, и ответственность за его конкретизацию лежит на механизме сопоставления.

3. Логическое программирование базируется на машине вывода, которая автоматически ищет ответы.
4. Функциональное программирование использует объекты более высокого уровня абстракции, потому что функции и типы можно использовать в качестве данных, а логическое программирование ограничено обычными типами данных.

Сферой применения логического и функционального программирования являются задачи искусственного интеллекта. В работах по искусственному интеллекту приходится оперировать сложными структурами данных и такими же сложными алгоритмами. Использование функционального стиля программирования естественно для приложений, характеризующихся многократным получением сложных структур данных из других таких же структур. Логика прямо используется для представления знаний и рассуждений и является средством их анализа. Логический стиль программирования – прекрасное средство для быстрого создания различных экспертных систем и интеллектуальных баз знаний, требующих сложной структуры запроса.

Список использованных источников

1. *Адаменко А. Н., Кучуков А. М.* Логическое программирование и Visual Prolog. – СПб.: БХВ-Петербург, 2003.
2. *Барендрегт Х.* Ламбда-исчисление. Его синтаксис и семантика. / Пер. с англ. – М.: Мир, 1985.
3. *Баррон Д.* Рекурсивные методы в программировании. / Пер. с англ. – М.: Мир, 1974.
4. *Бен-Ари М.* Языки программирования. Практический сравнительный анализ. / Пер. с англ. – М.: Мир, 2000.
5. *Бердж В.* Методы рекурсивного программирования. / Пер. англ. – М.: Машиностроение, 1983.
6. *Братко И.* Алгоритмы искусственного интеллекта на языке Prolog. / Пер. с англ. – 2-е изд. – М.: Вильямс, 2004.
7. *Братко И.* Программирование на языке Пролог для искусственного интеллекта. / Пер. с англ. – М.: Мир, 1990.
8. *Бузук Г.Л.* Логика и компьютер. – М.: Финансы и статистика, 1995.
9. *Гаврилова Т. А., Хорошевский В. Ф.* Базы знаний интеллектуальных систем. – СПб.: Питер, 2000.
10. *Городняя Л. В.* Основы функционального программирования. Курс лекций. Учебное пособие. – М.: ИНТУИТ.РУ, 2004.
11. *Дейт К.* Введение в системы баз данных. / Пер. с англ. – 6-е изд. – К.; М.; СПб.: Вильямс, 1999.
12. *Доорс Дж., Рейблейн А.Р., Вадера С.* Пролог – язык программирования будущего. / Пер. с англ. – М.: Финансы и статистика, 1990.
13. *Ин Ц., Соломон Д.* Использование Турбо-Пролога. / Пер. с англ. – М.: Мир, 1993.
14. *Керов Л.А. и др.* Экспертные системы: Инструментальные средства разработки: Учебное пособие. – СПб.: Политехника, 1996.
15. *Клоксин У., Меллиш К.* Программирование на языке Пролог. / Пер. с англ. – М.: Мир, 1987.
16. *Лавров С.С.* Программирование. Математические основы, средства, теория. – СПб.: БХВ-Петербург, 2001.
17. *Малпас Дж.* Реляционный язык Пролог и его применение. / Пер. с англ. – М.: Наука, 1990.
18. *Марселлус Д.* Программирование экспертных систем на Турбо Прологе. / Пер. с англ. – М.: Финансы и статистика, 1994.

19. *Новиков Ф.А.* Дискретная математика для программистов. – СПб.: Питер, 2002.

20. *Романовский И.В.* Дискретный анализ: Учебное пособие. – СПб.: Невский диалект, 1999.

21. *Смирнов В.М., Керов Л.А., Дерюшев В.А.* Создание Windows и Internet-приложений в виде виртуальных книг: Учебное пособие. – СПб.: ЭЛБИ, 1998.

22. *Стерлинг Л., Шапиро Э.* Искусство программирования на языке Пролог. / Пер. с англ. – М.: Мир, 1990.

23. *Тейз А., Грибомон П. и др.* Логический подход к искусственному интеллекту: от классической логики к логическому программированию. / Пер. с франц. – М.: Мир, 1990.

24. *Филд А., Харрисон П.* Функциональное программирование. – М.: Мир, 1993.

25. *Харрисон Дж.* Введение в функциональное программирование. / Пер. с англ., 1997.

26. *Хендерсон П.* Функциональное программирование. Применение и реализация. / Пер. с англ. – М.: Мир, 1983.

27. *Хоггер К.* Введение в логическое программирование. / Пер. с англ. – М.: Мир, 1988.

28. Язык Пролог в пятом поколении ЭВМ. // Сб. статей под ред. Ильинского. – М.: Мир, 1988.

29. *Довбуш Г. Ф.* Лабораторные работы по логическому программированию. Методические указания. – СПб: ПГУПС, 2005.