ЛОГИЧЕСКОЕ И ФУНКЦИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ

*Введение*

Целью логического и функционального программирования является вывод решений и они тесно связаны с задачами, решаемыми в искусственном интеллекте и экспертных системах (ЭС). На начальном этапе развития систем искусственного интеллекта (СИИ) и ЭС даже выделился целый класс специализированных языков программирования: языки логического и функционального программирования.

Языки программирования

Процедурные языки

Декларативные языки

Логические языки

Функциональные языки

Процедурная программа состоит из последовательности операторов и предложений, управляющих последовательностью их выполнения. В основе такого программирования лежат взятие значения какой-то переменной, совершение над ним действия и сохранение нового значения с помощью оператора присваивания, и так до тех пор пока не будет получено желаемое окончательное значение.

Функциональная программа состоит из совокупности определений функций. Функции, в свою очередь, представляют собой вызовы других функций и предложений, управляющих последовательностью вызовов. Каждый вызов возвращает некоторое значение и вызвавшую ее функцию, вычисление которой после этого продолжается. Этот процесс повторяется до тех пор, пока запустившая процесс функция не вернет результат пользователю.

В логических языках программирования для решения задачи достаточно описания структуры и условий этой задачи. Поскольку последовательность и способ выполнения программы не фиксируется, как при описании алгоритма, программы могут в принципе работать в обоих направлениях, то есть программа может как на основе исходных данных вычислить результаты, так и по результатам – исходные данные.

Наиболее известными языками функционального программирования являются ЛИСП и РЕФАЛ, а логического – Пролог. Однако, с развитием языков программирования (в частности, с появлением объектно-ориентированных языков) и баз данных область их применения сузилась. Так ЛИСП используется как оболочка Автокад, а РЕФАЛ как средство для построения метаязыков и метакомпиляторов.

Поэтому в дальнейшем внимание будет уделено не рассмотрению конкретных языков функционального и логического программирования, а подходам, лежащим в основе их реализации и являющимися базовыми при создании систем принятия решений.

ЛИСП и Пролог в свое время являлись базовыми для создания экспертных систем. Поэтому, для того чтобы наглядно представить какой круг задач решается с помощью логического и функционального программирования, рассмотрим задачи, возникающие в ЭС. Прежде всего, это задачи прямого и обратного вывода.

**Прямой и обратный вывод**

При использовании прямой цепочки рассуждений решается задача по известным условиям найти последствия. Обратная цепочка рассуждений применяется для того, чтобы по известным результатам найти причины их вызвавшие.

Такие задачи часто записывают в терминах продукционных систем представления знаний, в которых знания записываются в виде продукций/правил, имеющих вид:

**Если** <условие>, **То** <вывод>.

9. Должность - Отказ

10. Должность - администратор

7. Стаж работы

< 2

> 2

5. Средний балл за учебу

< 3.5

8. Должность - конструктор

> 3.5

2. Должность - Отказ

1. Имеет ученое звание

нет

4. Сделал открытия

да

6. Должность – научный сотрудник

да

3. Возможность принять

да

да

Рассмотрим сначала построение обратной цепочки рассуждений. Обратная цепочка рассуждений всегда начинается со следствия (часть **То** правила). Если в правилах, относящихся к проблемной области, не удается найти условную часть с выполняющимися условиями, необходимо ввести дополнительную информацию. Цепочка означает процедуру логической связи ряда правил.

Для представления таких задач принято использовать дерево решений - специальную диаграмму для представления возможных решений. Дерево решений состоит из вершин двух типов. Вершины решений, содержащие вопросы, обозначаются окружностями. Цели или логические выводы обозначаются прямоугольниками. Вершины нумеруются. Каждая вершина может иметь не более одного входа. Рассмотрим простейший пример с приемом на работу, часто используемый в литературе [1].

Дерево решений будем хранить в следующей таблице [2]:

*Таблица дерева решений.*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **№ вершины** | Переменная | **Значение** | **Исходная вершина** | **Дуга** | **Тип вершины** |
| 1 | Звание | - | - | - | решение |
| 2 | Должность | Отказ | 1 | нет | вывод |
| 3 | Возможность | да | 1 | да | вывод |
| 4 | Открытия | - | 1 | да | решение |
| 5 | Средний балл | - | 3 | да | решение |
| 6 | Должность | Научный сотрудник | 4 | да | вывод |
| 7 | стаж | - | 5 | < 3.5 | решение |
| 8 | должность | конструктор | 5 | > 3.5 | вывод |
| 9 | должность | Отказ | 7 | < 2 | вывод |
| 10 | должность | администратор | 7 | > 2 | вывод |

Для сохранения результатов будем использовать таблицу вывода (в начальный момент таблица пуста).

## Таблица вывода

|  |  |  |
| --- | --- | --- |
| **№ варианта** | **Переменная** | **Значение** |
|  |  |  |

### Алгоритм

1. Определить переменную логического вывода и ее значение.
2. Найти первое вхождение этой переменной в таблицу дерева решений с заданным значением и типом вершины «вывод». Если переменная не найдена – неудача. Установить переменную var = 1.
3. Выбрать исходную по отношению к полученной вершине вершину. Если ее нат, перейти к шагу 5. Если есть, записать в таблицу вывода новую строку со значениями полей **№ варианта = var, Переменная = Переменная из исходной вершины таблицы дерева решений, Значение = Дуга текущей вершины.**
4. Сделать исходную вершину текущей. Перейти к шагу 3.
5. Найти следующее вхождение переменной вывода в таблицу дерева решений. Если нет, конец, иначе var = var + 1. Перейти к шагу 3.

Пусть отказано в приеме на работу. Тогда в ходе выполнения алгоритма таблица вывода будет формироваться следующим образом.

Здесь пропущена таблица вывода на предпоследнем этапе.

Механизм, основанный на прямой цепочке рассуждений, функционирует следующим образом:

1. Вводится условие.

|  |  |  |
| --- | --- | --- |
| **№ варианта** | Переменная | Значение |
| 1 | Звание | нет |

|  |  |  |
| --- | --- | --- |
| **№ варианта** | Переменная | Значение |
| 1 | Звание | нет |
| 2 | Стаж | < 2 |

|  |  |  |
| --- | --- | --- |
| **№ варианта** | Переменная | Значение |
| 1 | Звание | нет |
| 2 | Стаж | < 2 |
| 2 | Средний балл | < 3.5 |

|  |  |  |
| --- | --- | --- |
| **№ варианта** | Переменная | Значение |
| 1 | Звание | нет |
| 2 | Стаж | < 2 |
| 2 | Средний балл | < 3.5 |
| 2 | Возможность | да |
| 2 | Звание | да |

1. Для каждой ситуации система ищет в базе данных (знаний) правила, в условной части которых содержится соответствующее условие.
2. В соответствии с констатирующей частью (частью ТО) каждое правило может генерировать новые ситуации, которые добавляются к уже существующим.
3. Система обрабатывает каждую вновь сгенерированную ситуацию. При наличии хотя бы одной такой ситуации выполняются действия, начиная с шага 2. Рассуждения заканчиваются, когда больше нет необработанных ситуаций

Алгоритм CLS

Для построения деревьев решений часто используется алгоритм CLS. Этот алгоритм циклически разбивает обучающие примеры на группы/классы в соответствии с переменной, имеющей наибольшую классифицирующую силу. Каждое подмножество примеров (объектов), выделяемое такой переменной, вновь разбивается на классы с использованием следующей переменной с наибольшей классифицирующей способностью и т.д. Разбиение заканчивается, когда в подмножестве оказываются объекты лишь одного класса. В ходе процесса образуется дерево решений. Пути движения по этому дереву с верхнего уровня на самые нижние определяют логические правила в виде цепочек конъюнкций.

Рассмотрим следующий пример. Проводится антропологический анализ лиц людей двух национальностей по 16 признакам.

Х1 (голова) – круглая – 1, овальная – 0.

Х2 (уши) – оттопыренные – 1, прижатые – 0.

Х3 (нос) – круглый –1, длинный – 0.

Х4 (глаза) – круглые – 1, узкие – 0.

Х5 (лоб) – с морщинами –1, без морщин – 0.

Х6(носогубная складка) – есть – 1, нет – 0.

Х7(губы) – толстые – 1, тонкие – 0.

Х8 (волосы) – есть – 1, нет – 0.

Х9(усы) – есть – 1, нет – 0.

Х10 (борода) – есть – 1, нет – 0.

Х11(очки) – есть – 1, нет – 0.

Х12(родинка) – есть – 1, нет – 0.

Х13(бабочка) – есть – 1, нет – 0.

Х14(брови) – поднятые вверх – 1, опущенные – 0.

Х15(серьга) – есть – 1, нет – 0.

Х16(трубка) – есть – 1, нет – 0.

Пусть имеется 16 объектов. Объекты с номерами 1 – 8 относятся к первому классу, 9 – 16 ко второму классу. Далее приводится таблица со значениями признаков для этих объектов.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| *№* | X1 | X2 | X3 | *X4* | *X5* | *X6* | *X7* | *X8* | *X9* | *X10* | *X11* | *X12* | X13 | *X14* | *X15* | *X16* | *Кл.* |
| 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
| 2 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 |
| 3 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 |
| 4 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 |
| 5 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 |
| 6 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 |
| 7 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| 8 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| 9 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 2 |
| 10 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 2 |
| 11 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 2 |
| 12 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 2 |
| 13 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 2 |
| 14 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 2 |
| 15 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 2 |
| 16 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 2 |

Объекты для этой таблицы (надо нарисовать).

Основное требование к математическому аппарату обнаружения закономерностей в данных заключается в интерпретации результатов. Правила, выражающие закономерности, формулируются на языке логических высказываний:

ЕСЛИ *А* ТО *В*,

ЕСЛИ (условие1) И (условие2) И … И (условиеN) ТО (условиеN+1),

где условиеi может быть *Xi =C1, Xi < C2, Xi > C3, C4 < Xi < C5* и т.д. Здесь *Xi -* переменная, *Cj –* константа.

Так классификация лиц в рассматриваемом примере может быть произведена с помощью четырех логических правил:

1. ЕСЛИ (голова овальная) И (есть носогубная складка) И (есть очки) И (есть трубка) ТО (класс1).
2. ЕСЛИ (глаза круглые) И (лоб без морщин) И (есть борода) И (есть серьга) ТО (класс1)
3. ЕСЛИ (нос круглый) И (лысый) И (есть усы) И (брови подняты вверх) ТО (класс2).
4. ЕСЛИ (оттопыренные уши) И (толстые губы) И (нет родинки) И (есть бабочка) ТО (класс2).

Математическая запись этих правил выглядит следующим образом:



Такие правила имеют две основных характеристики: точность и полноту.

Точность правила – это доля случаев, когда правило подтверждается, среди всех случаев его применения (доля случаев *В* среди случаев *А*).

Полнота – это доля случаев, когда правило подтверждается, среди всех случаев, когда имеет место объяснимый исход (доля случаев *А* среди случаев *В*).

Правила могут иметь какие угодно сочетания точности и полноты. За исключением одного случая, если точность равна нулю, то равна нулю и полнота, и наоборот.

Точное, но неполное правило: Люди смертны (*А* – человек, *В –* смертен).

Неточное, но полное правило: Студенты посещают занятия (*А –* студент, *В* - посещает).

Методы поиска логических закономерностей в данных обращаются к информации не только в отдельных признаках, но и в сочетании признаков. Это основное преимущество этих методов перед многими другими методами в ряде случаев.

Вернемся к примеру.

На первом шаге определяется признак с наибольшей дискриминирующей силой. Для этого определяется отношение вхождения объектов в разные классы в соответствии со значениями разных признаков:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Признаки | Х1 | Х2 | Х3 | Х4 | Х5 | Х6 | Х7 | Х8 | Х9 | Х10 | Х11 | Х12 | Х13 | Х14 | Х15 | Х16 |
| Кл1/Кл2 | 3/3 | 4/6 | 4/6 | 5/5 | 3/3 | 6/4 | 4/6 | 3/3 | 5/5 | 6/4 | 6/4 | 3/3 | 5/5 | 4/6 | 4/6 | 5/5 |

Здесь одинаковой и максимальной силой обладают сразу семь признаков: Х2, Х3, Х6, Х7, Х10, Х11, Х14, Х15. Поэтому случайным образом выбираем один из них в качестве ведущего. Пусть это будет Х6. От этого признака отходит две ветви. Первая для значения Х6 = 0, а вторая – для Х6 = 1.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Объекты | Признаки | | | | | | | | | | | | | | | |
|  | Х1 | Х2 | Х3 | Х4 | Х5 | Х6 | Х7 | Х8 | Х9 | Х10 | Х11 | Х12 | Х13 | Х14 | Х15 | Х16 |
| 2 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 |
| 7 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 |
| 12 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 |
| 13 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| 14 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 1 |
| 16 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 |
|  | 2/2 | 1/3 | 1/3 | 2/3 | 2/2 | 1/4 | 1/4 | 1/2 | 1/3 | 2/0 | 1/3 | 1/1 | 1/2 | 1/2 | 2/3 | 1/3 |

Для ветви Х6 = 0 окончательное решение дает признак Х10. Он принимает значение 1 на объектах 2 и 7 из первого класса и значение 0 на объектах 12, 13, 14 и 16 второго класса. Ветвь Х6 = 1 устроена значительно более сложно и требует дополнительных ветвлений. В результате получаем дерево.

Как следует из рисунка дерево логического вывода, выросшее из признака Х6 имеет 6 исходов. Только два из этих исходов имеет по четыре объекта (полнота 4/8). Один исход группирует три объекта (полнота 3/8), один – два объекта (полнота 2/8) и три исхода включают по одному объекту (полнота 1/8). Отсюда, качество алгоритма не очень высоко, так как обладает малой полнотой. Алгоритм, кроме того, способен приводить к качественным решениям только в случае независимых признаков.

# *Домашнее задание*

Построить дерево решений для проблемной области, заданной преподавателем, и построить цепочки прямых и обратных рассуждений для различных ситуаций.

X6

складка

нет

да

2,7/12, 13, 14, 16

X10

борода

0

1

2, 7

12, 13, 14, 16

1, 3, 4, 5, 6, 8/ 9, 10, 11, 15

X11

очки

0

5/ 9, 11, 15

0

1

X15

серьга

5

0

9, 11, 15

1, 3, 4, 6, 8/ 10

X16

трубка

1

1, 3, 4, 6

1

X2

уши

8/ 10

0

1

8

10

0

1

#### Компьютерное задание

Реализовать дерево решений, прямой и обратный вывод средствами Access. Для реализации необходимо использовать VBA. В Access 2000 по умолчанию установлена модель данных ADO (ActiveX Data Objects). Для установки MSDE, что соответствует модели данных DAO, используемой в Access 97, необходимо использовать другой файл установки. Необходимо вставить установочный компакт диск и дважды щелкнуть на имени файла SETUPSQL.EXE в папке \Sql\x86\Setup.

Иерархия ADO проще иерархии объектов модели

Объекты ADO предназначены для организации доступа к источникам данных, их редактирования и обновления. Модель ADO включает в себя объекты, необходимые для выполнения следующих задач:

1. Соединение с источником данных.
2. Создание объекта, реализующего команды SQL.
3. Указание столбцов, таблиц и значений в качестве переменных параметров в команде SQL.
4. Выполнение команды SQL.
5. Сохранение результатов выполнения в хеше.
6. Создание виртуального представления в хеше, чтобы пользователь мог сортировать, фильтровать данные в БД и перемещаться по ней.
7. Редактирование данных.
8. Обновление источника данных в соответствии со всеми изменениями сделанными в хеше.
9. Фиксация или отмена изменений, внесенных в ходе транзакции, и последующее закрытие транзакции.

К классам объектов в модели ADO относятся:

* Connection – представляет среду, в которой будет выполняться обмен данными с источником данных. Соединение должно быть установлено до начала выполнения любых других операций.
* Command – способ управления источником данных. Можно удалять, добавлять, обновлять и считывать данные из источника.
* Parameter – представляет переменные компоненты объекта Command. В командах часто необходимо указывать вспомогательные параметры, уточняющие способ выполнения команд. Параметры являются изменяемыми, так что перед выполнением команд их можно модифицировать
* Recordset – служит локальным хешем данных, считанных из источника данных.
* Field – представляет столбец таблицы Recordset. Поле содержит свойства определяющие поле. Пример таких свойств – Type, Value.
* Error – возвращает результат всякий раз, когда в приложении возникает ошибка. Каждый объект Connection имеет отдельное семейство объектов Error.
* Property – определяет объекты Connection, Command, Field, Recordset. Каждый объект ADO обладает набором свойств, задающим объект и управляющим его поведением.
* Collection – служит для объединения сходных объектов в группы.

Обращение к объектам ADO выглядит так:

ADODB. имя\_объекта.

При создании нового проекта, Access 2000 загружает только библиотеку объектов ADO. Если необходимо работать с DAO, добавляется библиотека объектов DAO в диалоге Preferences редактора VB. Для открытия VB Editor надо нажать Alt + F11. Диалог Preferences открывается командой меню Tools>References. В этом диалоге надо выбрать DAO 3.6 Object Library.

Для того чтобы связать объект Recordset в модели ADO с данными необходимо:

Dim rst As New ADODB.Recordset

rst.Open SQLVar,CurrentProject.Connection

Здесь SQLVar символьная переменная, в которой определяется набор данных либо как выражение SQL, либо как имя таблицы. Например, если необходимо открыть таблицу с именем Student, вторая строка будет выглядеть:

rst.Open “Student”, CurrentProject.Connection

В случае DAO необходимо создать объектную переменную rst типа Recordset без ADODB, а затем использовать метод OpenRecordset:

Set rst = CurrentDB.OpenRecordset(SQLVar, dbOpenDynaset).

Здесь необходимо быть аккуратным, поскольку написание для объектов Recordset в обеих моделях одинаково.

Для перехода в обеих моделях используются методы Move:

* rst.MoveFirst | MoveLast | MoveNext | MovePrevious | Move n – соответственно : Перейти к первой записи | к последней | к следующей | к предыдущей | на n записей

Метод Find используется при поиске в наборе записей, удовлетворяющих тем или иным условиям.

Переменная\_Recordset критерий, Пропустить Строки, Направление Поиска, Старт

Здесь:

* критерий – строковое значение (обязательно в кавычках), определяющее имя столбца (поля), оператор сравнения и искомое значение. Это единственный обязательный параметр.
* Пропустить строки – обозначает число строк, начиная с текущей или стартовой позиции, которое необходимо пропустить перед началом поиска.
* Направление поиска определяет должен ли поиск вестись по направлению к концу набора (adSearchForward) или к началу (adSearchBackward).
* Старт – закладка, обозначающая начальное положение указателя текущей записи при поиске: adBookmarkFirst (1) – первая запись, adBookmarkLast (2) – последняя запись, adBookmarkCurrent (0) – текущая запись.

Dim Rst As New ADODB.Recordset

Rst.Open “Student”, CurrentProject.Connection

Rst.Find “Sgroup = ‘АП51’”

Rst.Close

Значение критерия может быть строкой, числом или датой. Если значение имеет тип даты, то оно заключается в #, например, #11/11/03#.

При обновлении записей с помощью Recordset.Open необходимо установить значения нескольких свойств, определяющих набор данных. Самыми важными из этих свойств являются свойства LockType и CursorType.

LockType определяет право доступа к набору и принимает значения:

* AdLockReadOnly – объект доступен только для чтения (значение по умолчанию).
* AdLockPessimistic – записи блокируются сразу после начала редактирования по одной.
* AdLockOptimistic – устанавливает блокировку при вызове метода Update (используйте этот вариант).
* AdLockBatchOptimistic – разрешает пакетное обновление.

Свойство CursorType определяет тип курсора, применяемый в наборе данных. Его действие подобно определению набора данных в модели DAO. CursorType может принимать одноиз следующих значений:

* AdOpenForwardOnly – набор представляет собой статическую копию данных, пригодную для поиска, но поиск возможен только в направлении к концу набора (значение по умолчанию).
* AdOpenKeySet – позволяет вносить изменения в набор данных, но пользователь видит изменения, внесенные им самим.
* AdOpenDynamic – позволяет вносить изменения. Пользователь видит все результаты изменений. Наименее эффективен, но имеет больше всего возможностей. Поэтому используйте его.
* AdOpenStatic – набор представляет собой статическую копию данных.

Редактирование:

Rst.Open “Student”, CurrentProject.Connection, adOpenDynamic, adLockOptimistic

Rst.MoveFirst

Rst(“YearEnter”) = 2001

Rst.Update

Rst.Close

Обновляется поле YearEnter первой записи.

Добавление записи:

Rst.Open “Student”, CurrentProject.Connection, adOpenDynamic, adLockOptimistic

Rst.AddNew

Rst(“FIO”) = “Петров И.И.”

Rst(“YearEnter”) = 2003

Rst.Update

Ret.Close

Удаление записи:

Rst.Open “Student”, CurrentProject.Connection, adOpenDynamic, adLockOptimistic

Rst.MoveFirst

Rst.Delete adAffectCurrent

Rst.Update

Rst.Close

**2. Математические основы**

Математическими основами индуктивного и дедуктивного вывода являются математическая логика и ее развитие логика предикатов первого порядка.

Логика используется для того, чтобы доказывать правильность или неправильность тех или иных утверждений или высказываний. Это имеет большое значение при решении задач проектирования и для других вычислительных задач. Так искусственный интеллект во многом опирается на гипотезу символьных систем. Упрощенно эта гипотеза утверждает, что, организовав огромную структуру взаимосвязанных символов, представляющих реальные знания и сложный набор символьных процессов, позволяющий оперировать структурами с целью создания новых структур, можно создать машину, работающую также хорошо в смысле интеллектуальной деятельности, как человек.

**2.1 Алгебра высказываний**

Одним из основных понятий логики является понятие высказывания, правильность или неправильность которого мы стараемся определить. Попытаемся определить смысл этого термина. Высказыванием называют предложения, про которые разумно говорить, разумно считать, что они являются истинными или ложными. Неуточненность понятий «истинно» и «ложно» делают понятие высказывания расплывчатым. Однако, вводя ограничения можно это понятие уточнить. Фразы «Пойдем в кино?», «Да здравствует президент!» не являются высказываниями (как и любые вопросительные и восклицательные предложения). Фраза «Треугольник называется равносторонним, если все его стороны равны» (как и любое другое определение) также не является высказыванием. Фразы «2\*2 = 4» и «3>5» - высказывания (первое истинное, второе ложное). Фраза «В повести “Шинель” 200755 букв» - высказывание, но нам неизвестна его истинность. Фразу «Эта книга хорошая» не следует относить к высказываниям в традиционной логике в силу неопределенности понятия «хороший».

Поэтому, определим, как простое высказывание – высказывание, для которого в определенных условиях времени и места можно делать вывод об его истинности или ложности, причем это высказывание задается простым предложением («Сегодня идет дождь»).

Из простых высказываний с помощью связок можно строить сложные высказывания, и тогда возникает задача определения их истинности в зависимости от истинности простых высказываний, его составляющих.

Переменную, в область значений которой входят только высказывания (а точнее истинностные значения T, F) будем называть высказывательной переменной (двоичной или булевой). В силу определения областью значений этих переменных является двоичный алфавит.

Функции от любого конечного числа двоичных переменных, также способные принимать лишь два значения T и F,принято называть булевыми функциями. Иногда такие функции называют переключательными, так как реализующие такие функции схемы осуществляют переключение входных каналов, подавая на них сигналы, то одного, то другого вида.

Областью определения булевых функций от n – переменных служит совокупность всевозможных n – мерных упорядоченных наборов (векторов размерностью n), компонентами которых являются буквы двоичного алфавита T и F.

Для любого n = 1, 2, . . . среди n – мерных наборов можно ввести естественную лексикографическую упорядоченность. Для этого поставим в соответствие с F – 0, а с T – 1. Тогда набор преобразуется в последовательность 0 и 1. Такой набор уже можно рассматривать как представление целого неотрицательного числа в двоичной системе счисления. Например, TTTF → 1110 → 14. Это число будем называть номером набора. Эти наборы называются также кортежами или, используя геометрическую терминологию, точками.

Последнее название связано с тем, что имеется естественная возможность отождествлять различные n – мерные наборы с вершинами n-мерного куба.

Естественную упорядоченность наборов мы получим, если расположим их в порядке возрастания номеров. Первым в таком расположении является нулевой набор, все компоненты которого 0, последним – набор, все компоненты которого 1.

Отсюда, наборы размерности n нумеруются числами от 0 до 2n-1. А отсюда в частности вытекает:

1. Имеется в точности 2n двоичных n-мерных наборов.

Нетрудно также подсчитать число различных булевых функций от n переменных. Для каждого набора значений, независимо от значений других наборов, можно выбрать в качестве значений функции T или F. Следовательно, прибавление каждого нового набора к области определения булевой функции увеличивает число различных булевых функций ровно в два раза.

На одном наборе можно определить две различных булевых функции. На двух – 22 и т. д. Продолжая подобным образом и с учетом 1, получим:

2. Имеется точно  различных булевых функций от n переменных.

Следует отметить, что здесь и далее к числу функций от n переменных относятся и такие функции f(x1, x2, . . ., xn), которые не зависят от тех или иных переменных xi. В частности, в числе функций окажутся функции – константы (тождественная истина и тождественная ложь), которые можно рассматривать как функции от нуля переменных.

Условимся называть невырожденными функциями от n переменных такие функции, которые существенно зависят от всех этих переменных. Функции же от n переменных, сводящиеся к функциям от меньшего числа переменных называются вырожденными.

В теории булевых функций особое значение имеют функции одной и двух переменных. Имеется всего  = 4 разных функций одной переменной.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **x f** | **G1** | **G2** | **G3** | **G4** |
| **0** | 0 | 0 | 1 | 1 |
| **1** | 0 | 1 | 0 | 1 |

G1, G4 – константы 0 и 1.

G2 = x.

 и называется функцией отрицания или инверсии.

Число булевых функций от двух переменных равно . Выпишем сводную таблицу всех этих функций.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| x | y | F1 | F2 | F3 | F4 | F5 | F6 | F7 | F8 | F9 | F10 | F11 | F12 | F13 | F14 | F15 | F16 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |

Из выписанных функций шесть будут вырожденными, а именно функции F1, F4, F6, F11, F13, F16. Действительно, легко видеть:



Остальные функции будут невырожденными. Введем для них специальные названия и обозначения.

Функция F2 носит название конъюнкции или произведения или логического И. Для ее обозначения используется либо знак умножения, либо ∧. Отсюда:



Функция F7 носит название функции неравнозначности или суммы по модулю два. Для ее обозначения будем использовать:



Функция F8 носит название дизъюнкции или логическое ИЛИ. Для ее обозначения принято использовать знак ∨:



Функцию F9 будем называть отрицанием дизъюнкции или стрелкой Пирса, и обозначать через:



Функция F10 носит название функции равнозначности или эквивалентности и обозначается:



Функции F12 и F14 носят название импликации. Для их обозначения будем использовать:



Здесь следует отметить, что импликация соответствует высказыванию «Если А, то В». При этом возникает ситуация, что это высказывание с ложным А и истинным В истинно. Прежде всего, это соглашение, причем это соглашение ничему не противоречит. В повседневном языке утверждения с ложным А не употребляются.

Пример типа «Если бы я был космонавтом, я бы полетел на Луну» не опровергает наше утверждение. Здесь 1. Имеем дело не с «Если А, то В», а с «Если бы А, то бы В»; 2. Считать ложным такое утверждение не имеет смысла.

Возникает вопрос, можно ли бы было при ложном А, считать ложным высказывание «Если А, то В». В принципе можно, но математическая практика показывает, что выбранный нами вариант удобнее.

Приведем пример. Известно, что при возведении в квадрат обоих частей уравнения могут появиться новые корни. При этом подразумевается, что при возведении в квадрат корни потеряться не могут. Что это значит. Это значит, что любой корень уравнения:



является также корнем уравнения



Это высказывание мы считаем верным, хотя исходное уравнение может вовсе не иметь корней. Ясно, что здесь мы использовали введенное соглашение.

Функция F15 носит название отрицание конъюнкции или штрих Шеффера. Для ее обозначения используется:



Оставшиеся функции F3 и F5 назовем отрицание импликации:



Для построения исчисления высказываний важным является вопрос о построении функционально полной системы функций.

Будем говорить, что система логических функций называется функционально полной, если существует алгоритм, позволяющий строить из этих функций любые наперед заданные функции. Функционально полная система называется несократимой, если исключение какой-либо входящей в нее функции лишает систему свойства функциональной полноты.

Можно показать, что: Максимальное число функций в несократимой функционально полной системе булевых функций от двух переменных равно четырем. Таким образом, для булевой алгебры можно построить несколько вариантов функционально полных систем. В частности, стрелка Пирса и штрих Шеффера каждый по отдельности составляют функционально полную систему. Однако наиболее часто в качестве функционально полной системы используются: отрицание, конъюнкция и дизъюнкция.

Функции F7, F9, F15, F5, F3 уже выражены через эти операции. Выразим функции F10, F12, F14.



Фактически значение любой функции можно получить по таблице ее истинности. Однако, это значение можно получить, представив функции булевой алгебры с помощью алгебраических функций.

Алгебраически перечисленные выше операции можно выразить следующим образом:

*Отрицание (дополнение): *

*Конъюнкция: *

*Дизъюнкция: *

Тогда:

*Сумма по модулю 2: *

*Стрелка Пирса: *

*Эквивалентность: *

*Импликация: *

*Штрих Шеффера: *

*Отрицания импликации: *

Отметим, что эти формулы справедливы для бесконечнозначных и многозначных логик.

Законы булевой алгебры.

Для удобства разобьем законы на четыре группы.

*Первая группа.*

1.  (закон коммутативности для дизъюнкции).

2. 

3.  (первый закон поглощения).

4.  (второй закон дистрибутивности).

5.  (закон идемпотентности для дизъюнкции).

Следующие пять законов получаются заменой ∧ на ∨ и наоборот.

1′.  (закон коммутативности для конъюнкции).

2′.  (закон ассоциативности для конъюнкции).

3′.  (второй закон поглощения).

4′.  (первый закон дистрибутивности)

5′.  (закон идемпотентности для конъюнкции).

Каждый из законов 1′ - 5′ называется двойственным к соответствующему закону 1 – 5.

##### Вторая группа

1.  

2.  

3.  

4.  

5.  

6.  

##### Третья группа

1.  (закон двойного отрицания).

2. 

3. 

##### Четвертая группа

1. 

2.  (закон контрапозиции).

3. 

Сформулируем некоторые полезные следствия из приведенных законов.

С1. Выбрасывая из произвольной дизъюнкции дизъюнктивные элементы равные нулю, мы не изменим величину этой дизъюнкции.

С2. Если в дизъюнкции хотя бы один из элементов равен 1, то вся дизъюнкция равна 1.

С3. Выбрасывая из произвольной конъюнкции все сомножители равные 1, мы не изменим ее величины.

С4. Если в конъюнкции хотя бы один сомножитель равен 0, то все произведение равно 0.

С5. Дизъюнкция или произведение любого числа одинаковых элементов равняется А.

Эти следствия можно доказать по индукции.

С6. Если А(а1, . . ., ап) произвольное выражение булевой алгебры, построенное из выражений а1, . . ., ап с помощью операций отрицания, дизъюнкции и конъюнкции, то отрицание этого выражения равняется , где В(а1,…,ап) получается из А с помощью замены всех умножений на дизъюнкции, а всех дизъюнкций на умножения, при условии сохранения всех имевшихся в А отрицаний.

С7. Если к некоторому выражению А булевой алгебры применено более чем одно отрицание, то, не меняя значения выражения, можно исключить любое четное число отрицаний.

### Нормальные формы

Элементарной дизъюнкцией называется выражение,представляющее собой дизъюнкцию любого конечного множества попарно различных между собой переменных, часть которых (возможна пустая) взята со знаком отрицания. К числу элементарных дизъюнкций будем также относить выражения, состоящие из одной переменной (с отрицанием или без), а также константу 0.

В силу определения: 

являются элементарными дизъюнкциями, а  не являются.

Элементарным произведением называется выражение, представляющее собой произведение любого конечного множества попарно различных между собой переменных, над частью которых (быть может пустой) поставлен знак отрицания. К числу элементарных произведений также относятся выражения с одной переменной с отрицанием или без, а также константа 1. Элементарные произведения:  Неэлементарные произведения:  Переменные и их отрицания называются первичными термами или литералами.

Дизъюнкция любого числа первичных термов равна либо 1, либо элементарной дизъюнкции. Произведение любого числа первичных термов равно либо 0, либо элементарному произведению.

Нормальной дизъюнктивной формой (ДНФ) называется дизъюнкция любого конечного множества попарно различных произведений.

Конъюнктивной нормальной формой (КНФ) называется произведение любого конечного множества попарно различных дизъюнкций.

Алгоритм приведения к ДНФ и КНФ заключается в следующем.

1. Преобразовать выражение в соответствии с операциями отрицания.
2. Использовать первый (ДНФ) или второй (КНФ) законы дистрибутивности.

Пример: Привести к ДНФ и КНФ выражение.



На первом этапе обрабатываем знаки отрицания:



Раскрывая скобки, приведем к ДНФ:



При приведении к КНФ последовательно применяем второй закон к первой скобке выражения 



Рассмотрим некоторое множество М булевых переменных. В качестве такого множества будем выбирать множество аргументов той или иной булевой функции.

Элементарные дизъюнкции называются конституантами 0 для данного множества М булевых переменных, если они содержат в прямом либо инверсном виде все переменные множества М.

Элементарные конъюнкции называются конституантами 1 для данного множества М булевых переменных, если они содержат в прямом либо инверсном виде все переменные множества М.

Для переменных  произвольную конституанту нуля можно представить в виде  а произвольную конституанту 0 в виде  где  означает либо , либо  . Условимся нумеровать конституанты нуля и единицы с помощью тех же номеров, что и соответствующие им наборы переменных.

ДНФ/КНФ называется совершенной (СДНФ/СКНФ), если все составляющие ее элементарные произведения/дизъюнкции являются конституантами единицы/нуля для одного и того же множества М переменных. СДНФ/СКНФ называется СДНФ/СКНФ булевой функции f, если она равна этой функции и если множество, составляющих ее переменных, совпадает с множеством аргументов функции f.

Справедливо следующее: Любая булева функция имеет одну и только одну СДНФ/СКНФ.

Рассмотри процесс приведения к СДНФ/СКНФ.

Рассмотрим произвольную ДНФ ϕ от переменных x1, . . ., xn. Пусть некоторые элементарные произведения p, входящие в эту форму, не содержат какой либо переменной xj. Тогда эти произведения можно заменить равными им выражениями



Продолжая этот процесс относительно других переменных множества аргументов функции, не входящих в то или иное элементарное произведение, после повторения этой процедуры некоторое конечное число раз получим СДНФ выражения ϕ, поскольку в каждое составляющее ее элементарное произведение будут входить все переменные из множества аргументов функции.

Назовем этот процесс приведением ДНФ к совершенному виду.

Аналогичным образом можно построить процесс приведения КНФ к совершенному виду. Для этой цели к входящей в КНФ произвольной элементарной дизъюнкции q, не содержащей, например, переменной , добавляется равный нулю элемент . Затем к полученному выражению применяется второй дистрибутивный закон:



Продолжая по аналогии, мы сможем в каждую элементарную дизъюнкцию ввести все недостающие в ней переменные, после чего форма превратится в совершенную.

Пример: Выражение  привести к СДНФ и СКНФ.

1. 

2.



Синтез логических выражений.

Используя таблицу истинности любой логической формулы, можно определить ее в СДНФ или СКНФ. Для построения СДНФ в таблице истинности необходимо выбрать строки, в которых функция принимает значение 1 и сформировать конституанту 0. Переменная будет находиться в этой конституанте без знака отрицания, если она принимает значение 1 в этой строке и с отрицанием в противном случае. Соединить полученные конституанты знаком дизъюнкции.

Для получения СКНФ ищутся строки, в которых функция принимает значение 0. Строятся конституанты 1. Переменная берется со знаком отрицания, если она равна 1 и наоборот. Конституанты соединяются знаком конъюнкции.

Пример: Дана таблица истинности. Построить СДНФ и СКНФ.

|  |  |  |  |
| --- | --- | --- | --- |
| **x** | **y** | **z** | **f** |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 |

СДНФ: 

СКНФ: 

Минимизация булевых функции

Под минимизацией булевых функций понимается нахождение наиболее простого представления этой функции в виде суперпозиции функций, составляющих какую-нибудь фиксированную функционально полую систему S булевых функций. Наиболее простым обычно считается представление, содержащее наименьшее возможное число суперпозиций. При решении задачи минимизации важную роль играет понятие импликанты.

Булева функция  называется импликантой функции , если на любом значении переменных , на котором значение g равно 1, значение f также равно 1.

Простой импликантой функции f называется элементарное произведение , являющееся импликантой f и такое, что никакая его собственная часть (то есть произведение, получаемое из g отбрасыванием одного или нескольких компонент) уже не является импликантой функции f.

Дизъюнкция любого множества импликант одной и той же функции является импликантой этой функции.

Дизъюнкция всех простых импликант булевой функции совпадает с этой функцией и называется сокращенной дизъюнктивной нормальной формой.

Сокращенная нормальная форма является более экономной, чем СДНФ. Однако часто допускает дальнейшее упрощение за счет того, что некоторые из простых импликант могут поглощаться дизъюнкцией других простых импликант. Например, в сокращенной ДНФ простая импликанта yz поглощается дизъюнкцией остальных элементов формы: .

Однако справедливо следующее утверждение для сокращенной дизъюнктивной формы. Если сокращенная ДНФ не содержит никакой переменной, входящей в нее одновременно с отрицанием и без него, то эта форма является минимальной дизъюнктивной формой.

Приведение к минимальной нормальной форме от сокращенной ДНФ можно осуществит с помощью импликантной таблицы. Импликантная таблица представляет собой прямоугольную таблицу, строки которой обозначаются различными простыми импликантами, а столбцы конституантами единицы, на которых функция обращается в единицу.

На пересечении p-й строки k-го столбца импликантной таблицы ставится \* тогда и только тогда, когда импликанта составляет некоторую часть конституанты k. Для примера:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |
|  | \* | \* |  |  |  |
|  |  |  | \* | \* |  |
|  |  |  |  | \* | \* |
|  |  | \* |  |  | \* |

Система S простых импликант булевых функций f называется приведенной, если эта система полна и никакая ее часть не является полной системой импликант функции f. Дизъюнкция всех простых импликант, составляющих S, называется приведенной или тупиковой дизъюнктивной нормальной формой. Всякая минимальная ДНФ является тупиковой ДНФ.

Выделение приведенной системы простых импликант может быть проведено на основе импликантной таблицы. Для этой цели надо выбрать минимальные системы строк таблицы так, чтобы для каждого столбца среди выбранных строк нашлась хотя бы одна строка содержащая звездочку. Этот метод является методом перебора и практически применим для простых импликантных таблиц. В случае сложных таблиц можно применять алгебраический метод Петрика.

Суть этого метода заключается в том, что по импликантной таблице строится некоторое выражение, называемое конъюнктивным представлением этой таблицы. Для этого производится обозначение всех простых импликант различными буквами (например, A, B, C, …). После этого для каждого столбца α импликантной таблицы строится дизъюнкция



всех букв, обозначающих строки, на пересечении которых со столбцом α стоит \*. Беря произведение полученных qα для всех столбцов, конъюнктивное представление таблицы. Обозначим для нашего примера : . Тогда получим следующее представление таблицы:



Если в конъюнктивном представлении раскрыть все скобки в соответствии с законом дистрибутивности, получим дизъюнктивное представление.

Простые импликанты, символы которых в любой фиксированный терм дизъюнктивного представления составляют полную систему простых импликант функции.

Выполняя в дизъюнктивном представлении импликантной таблицы все элементарные поглощения  и устраняя повторения в соответствии с тождествами АА=А и А∨А = А, приходим к приведенному дизъюнктивному представлению импликантной таблицы.

Термам этого представления соответствуют все приведенные системы простых импликант функции.

В примере получим:



То есть получим 2 приведенные системы простых импликант (A, B, C) и (A, B, D). Им соответствуют две тупиковые ДНФ исходной функции:



*Алгоритм Квайна.*

1.Минимизируемая булева функция f от произвольного числа n переменных записывается в СДНФ f0.

2.Начиная с f0, строим последовательность ДНФ f0, f1, . . ., fi, . . . до тех пор пока две какие либо ДНФ fk и fk+1 не совпадут между собой. Переход от формы fi к fi+1 по следующему правилу: в форме fi выполняются все операции неполного склеивания



Применимые к элементарным произведениям длины n = . После этого исключаются все те элементарные произведения длины , которые могут быть исключены на основании формулы элементарного поглощения:

.

3.Результатом алгоритма Квайна к функции f является заключительная ДНФ fk.

Для любой булевой функции f результатом применения алгоритма Квайна к СДНФ будет сокращенная ДНФ этой функции.

Пример:



Операцию неполного склеивания можно применить к 1 и 3 элементу формулы, к 1 и 2, а также к 1 и 4. В результате получим:



Применяя формулу поглощения, получим:



Поскольку операция неполного склеивания дальше неприменима, f1 – сокращенная ДНФ.

*Диаграмма Вейча*

Диаграммы Вейча фактически представляют собой другую запись таблиц истинности и в простейшем случае для булевых функций двух, трех и четырех переменных имеют вид:

|  |  |
| --- | --- |
|  |  |
|  |  |









|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |













|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |















Для осуществления минимизации в таблице необходимо выделить смежные элементы, которыми являются для матрицы A элементы  и . Причем, операция сложения выполняется по модулю n, где n – размерность матрицы. Таким образом, элементы из первой и последней строки, из первого и последнего столбца могут быть смежными. Затем выбираются группы смежных элементов, количество которых должно быть степенью двойки. Эти группы определяют импликанты. Причем количество сомножителей в импликанте определяется как , где n – число аргументов функции,  - степень двойки для группы элементов. При составлении импликанты исключаются переменные, для которых единица имеется и в части отрицания для переменной и в части, где отрицание отсутствует. Необходимо выбрать группы так, чтобы их количество было минимально, и все единицы в таблице входили бы в группы (выбираются группы с максимальным ). Полученные импликанты связываются знаком дизъюнкции.

Пример:

|  |  |  |  |
| --- | --- | --- | --- |
|  | 1 | 1 |  |
|  | 1 | 1 |  |
|  |  | 1 |  |
| 1 |  | 1 |  |



















F=

##### Синтаксис, семантика и правила вывода в исчислении высказываний

Синтаксис исчисления высказываний определяется правилами грамматики:

###### Предложение : = Элементарное предложение / Сложное предложение

*Сложное предложение : = Предложение Связка Предложение / ^Предложение / (Предложение)*

*Связка : = ∧ / ∨ / ^ / → / ≈*

Семантика исчисления высказываний определяется с помощью таблиц истинности.

К правилам вывода относятся:

1.

Если посылка А есть истина, то и заключение В есть истина.

2.Исключение И:



Знание того, что А и В есть истина, должно означать, что А есть истина и В есть истина.

3.Интродукция ИЛИ:



Если А есть истина, то А или В есть истина. То же самое имеет место, если В есть истина.

4.Интродукция И:



Если А есть истина и В есть истина, то А И В есть истина.

5.Двойное отрицание:



Если А есть не не истина, то А есть истина.

6.Единичная резолюция:



Если А или В есть истина и не В, то А есть истина. Точно также, если не А, то В – истина.

7.Резолюция:



Если А или В и не В или С, то, поскольку В не может быть истинно и ложно одновременно, должно быть А или С истинно.

Пример: Имеется следующая информация.

Если аккумулятор машины разряжен, то машина не заводится. Если машина Ивана не заводится и текущее время оказывается позже восьми часов утра, то Иван опоздает на поезд. Однажды после восьми утра аккумулятор Ивана оказался разряженным.

Используя логические правила вывода, доказать, что Иван опоздает на поезд.

В символьных обозначениях информация может быть представлена в следующем виде:

P: аккумулятор разряжен.

Q: машина не заводится.

R: время после восьми утра.

S: Иван опоздал на поезд.

Правило 1: P → Q.

Правило 2. Q∧R → S.

Известно, что P и R есть истина. Задачей является доказать S. Доказательство строится следующим образом:

1. P – дано.
2. R – дано.
3. Q следует из 1 и правила 1 по правилу modes ponens.
4. Q∧R следует из 3 и 2 по правилу интродукции И.
5. S следует из 4 и правила 2 по правилу modes ponens.

Исчисление предикатов предполагает, что мир можно моделировать с помощью фактов. Но для реальных приложений исчисления высказываний недостаточно.

*Практические задания*

Задание 1

Даны логические функции:



Получить значения этих функций с помощью таблиц истинности. Преобразовать к алгебраическому виду и определить значения для x =1, y =1, z = 0. Преобразовать к виду с использованием только операций дизъюнкции, конъюнкции и отрицания. Упростить. Привести к СДНФ и СКНФ. Восстановить СДНФ и СКНФ по таблице истинности. Построить импликантную таблицу и определить сокращенную ДНФ с использованием метода Петрика. Минимизировать с использованием алгоритма Квайна и диаграммы Вейча.

Задание 2

Если собака видит кошку, то она за ней гонится. Если за котом Васькой гонится собака и рядом есть дерево, то кот Васька забирается на дерево. В саду много деревьев. Однажды, в саду Ваську увидела собака.

Доказать, пользуясь логическими правилами вывода, что Васька забрался на дерево.

*Компьютерное задание*

Реализовать программу решающую следующую задачу:

На входе программы – таблица истинности для функции четырех переменных.

На выходе программы – СДНФ, СКНФ, диаграмма Вейча и минимальная нормальная форма.

**2.2 Исчисление предикатов**

# Исчисление предикатов расширяет язык исчисления высказываний так, что мир оказывается, состоящим из объектов, отношений и свойств.

Логику предикатов можно рассматривать как компоненту естественного языка, имеющую в соответствии со сложностью синтаксических правил иерархическую структуру, которую образуют предикаты первого порядка, второго и так далее. Для логики предикатов определено множество значений и на его основе определены слова как последовательности знаков. Функцией языка предикатов является задание слов двух типов:

1. Слова, задающие сущности изучаемого мира.
2. Слова, задающие атрибуты / свойства этих сущностей, а также их поведение и отношения.

Первый тип слов называется термами, второй – предикатами.

Некие сущности и переменные определяются упорядоченными последовательностями конечной длины из букв и символов, исключая зарезервированные. Константы и переменные определяют отдельные объекты рассматриваемого мира. Последовательность из n констант или переменных (1 ≤ n < ∞), заключенная в круглые скобки, следующие за символом функции, имя которой задано некоторой конечной последовательностью букв, называется функцией.

Например, функция f(x, y) принимает некоторые значения, которые определяются значениями констант и переменных (аргументов функции), содержащимися под знаком функции. Эти значения, так же как и аргументы, являются некоторыми сущностями рассматриваемого мира. Поэтому все они объединяются общим названием терм (константы, переменные, функции).

Атомарным предикатом (атомом) называется последовательность из n (1 ≤ n <∞) термов, заключенных в круглые скобки, следующие за предикатным символом, имя которого выражается конечной последовательностью букв. Предикат принимает одно из двух значений true или false в соответствии со значениями, входящих в него термов.

# Предикат ≅ Нераспространенное простое предложение

Из атомов с помощью, выполняющих функции союзов, символов составляются логические формулы, соответствующие сложным предложениям. В логике предикатов используются два класса символов. Первый класс соответствует союзам и включает операции дизъюнкции, конъюнкции, отрицания, импликации и эквивалентности.

Символы первого класса позволяют определять новый составной предикат, используя уже определенные предикаты. Различие между символами первого класса лежит в правилах, в соответствии с которыми определяются значения истинности или ложности составного предиката в зависимости от истинности или ложности элементарных предикатов. Символы → и ≈, вообще говоря избыточны так, как:



но используются т.к. → эквивалентен фразе «Если А, то В», а ≈ - «А и В эквивалентны».

В качестве символов второго класса используются ∀ и ∃. Эти символы называются кванторами общности и существования, соответственно. Переменная, которая квантифицирована, т.е. к ней применен один из кванторов , называется связанной. Квантор общности является обобщением, аналогом конъюнкции, а квантор существования – обобщением, аналогом дизъюнкции на произвольное, не обязательно конечное множество.

Действительно, пусть  Тогда для любого предиката U выполняется:



Аналогом законов Де Моргана для кванторов являются:



Часто возникает ситуация, когда некоторые переменные связываются кванторами ∀, а все другие - ∃. В этом случае может возникнуть неоднозначность при интерпретации кванторов.

Пусть задан некоторый предикат U(x, y). Очевидно, что возможно восемь случаев связывания его кванторами существования и общности:



Необходимо дать интерпретацию этим восьми случаям.

Рассмотрим, например, предикат *подсистема(x, y)*, который задает отношение *x подсистема y.* Пусть переменная x связана квантором общности, а y – квантором существования. В этом случае существует две интерпретации: 1. «Для всех x, существует y, для которых x подсистема», 2. «Существует y, что все x его подсистемы».

*1. (∀x)(∃y)подсистема(x, y)*

Подсистема x

Система y

*2. (∃y)(∀x)подсистема(x, y)*

Подсистема x

Система y

Порядок следования связанных квантором переменных определяется при чтении предиката слева направо. Дадим интерпретацию для других значимых случаев, которые можно выразить этим предикатом и кванторами:

* + *(∀x)(∃y)подсистема(x, y)* – все объекты являются подсистемами;
  + *(∃x)(∀y)подсистема(x,y)* – существует объект, который является подсистемой любого объекта;
  + *(∀y)(∃x)подсистема(x, y)* – для всякого объекта существует объект, являющийся его подсистемой;
  + *(∀x)(∃y)подсистема(x, y)* – существует объект, который является чей-то подсистемой.

Сделаем некоторые важные обобщения.

1.Чтобы найти отрицание выражения, начинающегося с кванторов, надо каждый квантор заменить на его двойственный и перенести знак отрицания за кванторы. Отсюда:



2.Одноименные кванторы можно переставлять. Разноименные кванторы можно переставлять только в одну сторону. Отсюда:



Если  то 

Действительно, если существует объект, который является чьей-то подсистемой, то для каждого объекта будет существовать объект, являющийся его подсистемой.

Если  то 

Действительно, если существует y, что все x его подсистемы, то для всех x существует y, для которого x подсистемы. Однако, перестановка в обратную сторону неверна. Например:

Если  то  необязательно.

Действительно, если для каждого объекта существует объект, являющийся его подсистемой, то это не означает, что существует объект, который является чьей-то подсистемой.

3.

Докажем эту равносильность.



В этой выкладке мы опирались на следующее утверждение:



Определение ППФ и семантика логики предикатов

Комбинируя два типа символов, можно рекурсивно определить составную формулу логики предикатов, называемую правильно построенной формулой (ППФ) или логической формулой.

1. Атомарный предикат является ППФ.
2. Если F и G являются ППФ, то  также ППФ.
3. Если F(x) – ППФ, то (∀x)F(x) и (∃x)F(x) – ППФ.
4. Все результаты, полученные применением конечного числа раз (1) – (3) являются ППФ. Ничто другое не является ППФ.

Формулы логики предикатов строятся безотносительно к понятиям задаваемой предметной области. Если решено, что этими формулами будет описываться конкретная предметная область, то должно быть установлено соответствие между понятиями предметной области и этими формулами. Это предполагает следующие действия:

1. Устанавливаются соответствия между константами логики предикатов и сущностями этой области. Константы ≡ Имена объектов.
2. Устанавливаются соответствия между формулами и функциональными отношениями предметной области.
3. Устанавливаются соответствия между атомарными предикатами и концептуальными отношениями предметной области.

Таким образом, в язык приносится конкретное смысловое содержание, то есть семантика логики предикатов. Обычно такая интерпретация формально представляется следующим образом:

1. Задается непустое множество D, определяющее сущности рассматриваемой предметной области, и элементы из D определяются как константы или переменные.
2. Для функций (функциональные отношения), определенных на множестве аргументов от  до D назначаются функциональные символы.
3. Каждому предикату n переменных назначается отношение, определенное на Dn, и его значение – True или False.

Множество D, рассматриваемое с позиций логики предикатов, называется областью переменных.

Значения ППФ оцениваются следующим образом:

1. Если известны значения логических формул F и G, то значения  оцениваются по таблице истинности.
2. Если для всех x∈M F оценивается как истина, то истиной является (∀x)F(x).
3. Если хотя бы для одного x∈M F оценивается как истина, то (∃x)F(x) тоже истина.

Предложения

Когда все переменные предиката являются связанными, то такой предикат называется предложением. Различие между ППФ, являющимися и не являющимися предложениями, состоит в том, что предложениям можно однозначно поставить в соответствие значение True или False, в то время как во втором случае нельзя непосредственно по виду формулы вынести суждение об ее истинности или ложности. Например, предикатная формула *подсистема (x, y)* не является предложением. Если в нее подставлены определенные значения, например, *x = процессор, y = ЭВМ*, то выражение *подсистема (процессор, ЭВМ)* принимает значение *True,* а при подстановке *x = человек,* выражение *подсистема(человек, ЭВМ)* является *False.* То есть истинность или ложность предикатной формулы можно оценить тогда и только тогда, когда в переменные подставлены некоторые конкретные сущности (в этом случае формула называется высказыванием). Отметим, что значение предикатных формул со связанными переменными можно определить, и не производя такого рода подстановки.

Например,  - у каждого человека имеется отец – является истиной, а  - любой является чьим-то отцом – является ложью.

Предположим, что имеется некоторое множество логических формул Σ. Если существует такая интерпретация, что все эти формулы принимают значение истина, то подобная интерпретация называется моделью. Например, рассмотрим множество:



Тогда интерпретация вида (*человек (Сократ) = True)* и (*смертен (Сократ) = True)* – является моделью так как все логические формулы Σ есть истина. Не обязательно, что такая модель единственна.

Пусть имеется некоторая группа Σ логических формул. Если для всех моделей Σ некоторая логическая формула σ есть истина, то принято говорить, что σ является логическим заключением (консеквентом) в Σ. Факт реализации логического консеквента записывается в виде Σ⏐=σ.

Правила вывода логики предикатов

Вывод представляет собой процедуру, которая из заданной группы выражений выводит, отличное от заданного выражение. Когда группа выражений, образующих посылку, является истиной, то должно гарантироваться, что выражение, выведенное из них в соответствии с правилом вывода, также является истиной.

В логике предикатов в качестве такого правила вывода используется правило, которое из двух выражений A и A→B выводит новое выражение B. Это правило называется правилом дедуктивного вывода.

Для описаний правил вывода во многих случаях используется нотация (как это указывалось выше), при которой над чертой записывается группа выражений, принимаемых за посылку, а под чертой – выражение, которое выводится:



Такой тип правила вывода носит название modus ponens.

Можно многократно использовать одно и тоже правило вывода. Например, если помимо выражений A, A→B существует выражение B→C, то можно вывести C, дважды использовав приведенное правило. Получение выражения σ применением конечного числа раз правила вывода к заданной группе выражений Σ будем записывать в виде: 

При этом говорят, что σ дедуктивно выводится из Σ. Очевидно, что из вышеуказанного, легко выводится еще одно правило:



Практические задания

*Задание 1.* Задан предикат **выполнение(x, y)** , который задает отношение «y является результатом выполнения программы x». Дать интерпретацию утверждений:

(∀x)(∃y)выполнение(x,y);

(∃x)(∀y) выполнение(x,y);

(∀x)(∀y) выполнение(x,y);

(∃x)(∀y) выполнение(x,y);

(∀y)(∃x) выполнение(x,y);

(∃x)(∃y) выполнение(x,y).

Задан предикат **реализация(x, y)**, который означает «программа x реализована на языке y». Записать утверждения:

А) Существует программа, реализованная на Паскале, имеющая в качестве результата 1000.

Б) Любая программа, написанная на C, дает результат.

**3. Функциональное программирование**

Функциональное программирование представляет собой модель индуктивного вывода, реализуемую с помощью рекурсивных процедур, λ-исчисления и списковых структур.

**3.1 Индуктивный вывод**

Индуктивный вывод – это вывод из имеющихся данных, объясняющего их общего правила. Например, пусть известно, что есть некоторая функция от одной переменной. Рассмотрим как выводится f(x), если последовательно задаются в качестве данных пары значений (1, f(0)), (1, f(1)). Вначале задается (0, 1) и имеет смысл ввести постоянную функцию f(x) = 1. Затем задается пара (1, 1), которая удовлетворяет f(x)=1. Следовательно, нет необходимости менять вывод. Пусть теперь задается (2, 3). Это не согласуется с нашим выводом. Предположим, что методом проб и ошибок удается найти . Она удовлетворяет заданным до сих пор фактам. Далее, если будут следовать факты (3, 7), (4, 13), (5, 21), …, убедимся в справедливости предшествующего вывода. Таким образом в индуктивном выводе в каждый момент времени объясняются все данные, полученные до этого момента. Если данные, позже, перестанут удовлетворять полученному выводу, то его придется менять. Следовательно, индуктивный вывод – это неограниченно долгий процесс.

Для определения индуктивного вывода необходимо уточнить:

1. Множество правил объектов вывода.
2. Метод представления правил.
3. Способ показа примеров.
4. Метод вывода.
5. Критерий правильности вывода.

В качестве правил – объектов вывода можно рассматривать:

* + *функции*;
  + грамматики языков;
  + программы.

Метод представления удобно организовывать в виде пар (x, f(x)), *последовательности действий, вычисляющих f(x)* и т.д.

Вывод реализуется благодаря неограниченному повторению процесса:

*Запрос входных данных → предположение → выходные данные.*

Критерием правильности вывода считается идентификация в пределе. Говорят, что машина вывода M идентифицирует в пределе правило R, если при показе примеров правилу R последовательность выходных данных, генерируемых M, сходится к некоторому представлению τ, а именно: все выходные данные, начиная с некоторого момента времени, совпадают с τ, при этом τ называют правильным представлением R. Кроме того, говорят, что множество правил G позволяет сделать индуктивный вывод, если существует некоторая машина вывода M, которая идентифицирует в пределе любое правило R из множества G.

Характерным примером индуктивного вывода является математическая индукция.

*3.1.1 Математическая индукция*

Методом математической индукции называются утверждения вида:



Пусть переменная n имеет область N (натуральные числа). Чтобы доказать первое утверждение, достаточно доказать два утверждения:

 (4)

 (5)

Первое из этих утверждений называется базисом индукции, второе - индукционным шагом. Для доказательства второго утверждения берут произвольное натуральное число, обозначают его какой-либо буквой, скажем k, и доказывают импликацию:

*U(k) → U(k+1)* (6)

Как следует из определения квантора общности, доказав это получим (5). Для доказательства (6) предполагают, что истинна посылка:

*U(k)* (7)

и выводят из этого предположения, что истинно ее заключение *U(k+1)*. Утверждение (7) называется предположением индукции.

При доказательстве утверждения (2) базисом индукции является утверждение *U(a),* индукционным шагом – *(∀n≥a)(U(n)→U(n+1))*, предположением индукции – утверждение *U(k)*, где k – произвольное натуральное число большее или равное a.

При доказательстве утверждения (3) базисом индукции является утверждение *U(a),* индукционным шагом – утверждение *(∀n:a≤n<b)(U(n)→U(n+1))*, предположением индукции *U(k),* где *k* – произвольное натуральное число такое, что *a≤n<b.* Такая индукция называется индукцией по интервалу. От индукции по интервалу можно перейти к индукции спуска. Индукцией спуска называется индукция, базисом которой является утверждение *U(b),* индукционным шагом – утверждение *(∀n:a≤n<b)(U(n+1)→U(n))*. Предположением индукции в этой форме является утверждение *U(k+1)*, где *k* – произвольное натуральное число такое, что *a≤n<b.*

Иногда утверждение 1 удобно доказывать возвратной индукцией. Возвратная индукция - это индукция с базисом *U(1)*, но с индукционным шагом *(∀n)(∀m≤n)(U(m) → U(n))*. Предположением индукции является *(∀m<k)U(m),* где *k* – произвольное натуральное число. Возвратная индукция с измененным базисом и индукционным шагом применяется и при доказательстве утверждений 2 и 3.

Для доказательства утверждений 1-3 часто бывает удобной индукция с двойным базисом, то есть для случая 1 индукция с базисом *U(1)∧U(2)* и индукционным шагом *(∀n) ((U(n)∧U(n+1))→U(n+2)).* Иногда удобно применять индукцию с тройным, четырехкратным и т.д. базисом.

Для доказательства утверждений вида

 (8)

применяется индукция по двум переменным. Однако, утверждение (8) часто удается свести к индукции по одной переменной. Для этого формируем в качестве значения переменной *m* произвольное число и обозначаем его *а.* Докажем утверждение *(∀n)U(a,n).* Из этого утверждения очевидно следует (8). Но последнее утверждение имеет вид 1 и его можно доказать индукцией. При такой схеме *n* называется индукционной переменной. Говорят также, что (8) доказывается по *n* при фиксированном *m.*



*3.1.1 Практические задания*

1.Требуется найти ошибку в доказательстве того, что натуральные числа равны между собой.

Пусть *A* переменная с областью определения *R(N)* (множества натуральных чисел), *n* – переменная с областью определения *N* (натуральные числа). Обозначим через *U(A, n)* высказывание: «Если множество *A* содержит *n* элементов, то в *A* нет двух неравных натуральных чисел». Очевидно, утверждение



равносильно утверждению задачи.

Докажем утверждение (1) индукцией по *n*, причем индукцию будем применять в ее простейшей форме. Базисом индукции является:

(*∀A)U(A,1). (2)*

Докажем (2). Возьмем произвольное *M⊂N* и докажем

*U(M,1). (*3)

Тем самым будет доказано утверждение (2). Но на основе определения *U*, (3) утверждает: «Если множество *M* содержит один элемент, то в *M* нет двух неравных натуральных чисел», что очевидно. Предположим теперь в качестве предположения индукции, что *(∀A)U(A,n)* верно для некоторого натурального *k*, то есть, что верно

*(∀A)U(A,k)* (4)

и докажем, что верно

*(∀A)U(A,k+1)* (5)

# Тем самым утверждение (1) будет доказано. Для доказательства (5) возьмем произвольное *M⊆N* и докажем

*U(M,k+1)* (6)

Тем самым (5) будет доказано. На основе определения *U*, (6) есть утверждение: «Если множество *M* содержит *k+1* элементов, то в *M* нет двух неравных натуральных чисел». Чтобы доказать эту импликацию, предположим, что ее посылка верна. Пронумеруем как-нибудь элементы множества *M.* Пусть, например:



Докажем, что в множестве *M* нет двух неравных натуральных чисел. Тем самым доказательство будет закончено. Рассмотрим два вспомогательных множества:



Каждое из них получено выбрасыванием из *M* одного элемента и, следовательно, содержит *k* элементов. В силу предположения индукции (4) *U(K,k)* – «Если множество *K* содержит *k* элементов, то в *K* нет двух неравных натуральных чисел». Но множество *K* как раз содержит *k* элементов, следовательно, в нем нет двух неравных натуральных чисел. Отсюда:

** (7)

Используем еще раз предположение индукции (4). Возьмем теперь в качестве значения *A* множество *L*. Теперь из (4) получим утверждение *U(L,k)*, что означает: «Если множество *L* содержит *k* элементов, то в нем нет двух неравных натуральных чисел». Но множество *L* содержит как раз *k* элементов, следовательно, в нем нет двух неравных натуральных чисел. В частности:

 (8)

Из (7) и (8) следует, что в *M* нет двух неравных натуральных чисел. Доказательство закончено.

**3.2 Рекурсия**

Особое место для систем функционального программирования приобретает рекурсия, поскольку она позволяет учитывать значения функции на предыдущих шагах.

С теоретической точки зрения рекурсивные определения являются теоретической основой всей современной теории вычислимых функций. Рассмотрим два способа вычисления *f(1)+f(2)+…+f(n).* При итерации сделаем следующим образом. Определим подпрограмму:

Sub Add(S,k,f)

S=S+f

k=k-1

End Sub

Тогда процедуру без использования цикла можно определить следующим образом:

S=0

k=n

I: Add(S,k,f(k))

J: If k≠0 Then Goto I

Здесь подпрограмма Add выполняется n раз последовательно и независимо, причем каждый раз используется только одна команда возврата. Это итерация.

Для рекурсии построим функцию:

If k=0 Then

Sum(f,k)=0

Else

Sum(f,k)=f(k)+Sum(f,k-1)

End If

Теперь достаточно просто узнать значение Sum(f,n). Рассмотрим частный случай при n=2. Из определения следует, что необходимо вычислить f(2), а затем обратиться к вычислению Sum(f,1), результат вычисления которого должен быть прибавлен к f(2). Следовательно, сохранить в памяти f(2), установить еще один возврат и обратиться еще один раз к нашему определению. Теперь вычисляем f(1), снова запоминаем результат в памяти, устанавливаем третий возврат и в третий раз обращаемся к определению для вычисления Sum(f,0). Последняя функция равна 0, и мы выходим из определения, используя возврат, установленный перед обращением к определению. Далее прибавляем 0 к f(1), снова выходим из определения, используя второй возврат, прибавляем 0+f(1) к f(2) и производим окончательный выход. Это рекурсия. Определение Sum использовалось не последовательно и независимо, а с вложением последующего использования в предыдущее (что характерно для индуктивного вывода), три команды возврата одновременно хранились в памяти и использовались по принципу «последний пришел – выполнился первый» (LIFO).

Здесь мы рассмотрели пример простой рекурсии. Другим более сложным примером рекурсии является вычисление чисел Фибоначчи. Их можно определить с помощью следующих формул:

F(n) = F(n – 1) + F(n – 2)

F(2) = 1

F(1) = 1

Формально число Фибоначчи можно вычислить по следующей явной формуле:

F(n) = [(1 + √5)n – (1 - √5)n] / (2n √5).

Эта формула относительно сложна. На самом деле в этом виде задача даже полностью не решена, поскольку алгоритм использования формулы требуется уточнить. Например, осуществлять ли раскрытие внутренних скобок по формуле бинома? Какое значение брать для числа √5, то есть сколько брать десятичных знаков? Очевидно, что хотя результат должен быть целым, он не будет таковым. Следовательно, встает вопрос, до какого соседнего целого нужно округлять? Как быть при этом уверенным в результате?

Для этой задачи можно использовать решение с непосредственной подстановкой:

Sub F(n)

If n > 2 Then

F(n) = F(n – 1) + F(n – 2)

Else

F(n) = 1

End If

End Sub

Для приведенных выше функций существуют алгебраические выражения, по которым их можно вычислить. Поэтому, используемые для них рекурсии, будем называть простыми. Однако, существуют функции, которые не являются простыми рекурсиями. Эти функции можно определить рекурсивно, но нельзя записать в терминах алгебраических выражений. Характерным примером является функция Аккермана. Пытаясь определить эту функцию алгебраически, получим только последовательность экспонент, записанных через многоточие. С другой стороны существует простая запись этой функции через рекурсию:

A(m,n)=iif(m=0,n+1,iif(n=0,A(m-1,1),A(m-1,A(m,n-1)))).

Вообще говоря, вычисление таких функций может быть бесконечным (характерная особенность индуктивного вывода). В качестве примера приведем функцию *f(m,n)*, результатом которой является 1 в случае, если в десятичной записи числа π встречается фрагмент из последовательности повторяющихся цифр *m* длиной *n.*

Можно показать, что алгоритм вычисления этой функции существует, но неизвестно каков он. Мы можем последовательно вычислять знаки π в надежде, что искомая последовательность обнаружится. Такие функции еще называются общерекурсивными.

В зависимости от того, как оформлен вызов рекурсии, можно выделить еще несколько ее разновидностей. Рекурсия называется параллельной, если она встречается одновременно в нескольких аргументах функции, то есть когда тело определения функции *f*, содержит вызов некоторой функции *g*, несколько аргументов которой являются рекурсивным вызовом *f:*

*f(…g(…,f,…,f,…)).*

Рекурсия является взаимной между двумя или более функциями, если они вызывают друг друга, то есть когда в определении функции *f* вызывается функция *g*, которая в свою очередь содержит вызов функции *g:*

*f((…)(…g…));*

*g((…)(…f…)).*

Рекурсия более высокого порядка является рекурсией, в которой аргументом рекурсивного вызова является рекурсивный вызов:

*f(…f(…f…)…).*

*3.2.1 Компьютерное задание*

Дана последовательность символов *a1, a2, …, an.* С применением рекурсии реализовать процедуру инверсии данной последовательности, то есть процедуру получения последовательности *b1, b2, …, bn* такой*,* что *b1 = an, b2 = an-1, …, bn-1 = a2, bn = a1.*

**3.3 λ-исчисление**

В алгебре приводится четкое различие между связанными и свободными переменными в контексте некоторых конструкций. Внутри этих конструкций свободные переменные имеют тот же смысл, что и вне них. Связанные же переменные полностью принадлежат самим конструкциям и вне них пусты, то есть могут быть заменены любыми буквами (за исключением тех, которые используются в данной конструкции).

В выражении



буква *x* может быть заменена любой другой буквой (за исключением *t* или *f*) и смысл выражения от этого не изменится в любом контексте, где это выражение может быть использовано.

В определении функции вида:

*…* где *g(x) = a sin(px + q)*

буква *x* также пуста.

С точки зрения вычислительной математики возникает проблема, является ли *x* именем объекта (областью рабочей памяти) или нет; или *x* – это объект, значением которого является другое имя. Этот вариант может иметь дальнейшее развитие, когда фактический параметр в *g(x)* является выражением отличным от простой переменной, например, как в записи *g(t2 + 2).*

Очевидно, что в основе лежит вопрос определения функций. Для этих целей в 1941 году Черчем было введено λ -исчисление. Задать функцию – это означает указать закон соответствия, приписывающий значение функции каждому допустимому значению аргумента. Пусть *M* – некоторая формула, содержащая *x* в качестве свободной переменной. Тогда *λx.[M]* есть функция, значение которой для любого аргумента получается подстановкой этого аргумента в *M* вместо *x*. Операцию образования выражения *λx.[M]* из *M* и *x* называют λ - операцией или функциональной абстракцией.

В λ - исчислении исследуются такие классы формальных систем, в которых используются общие способы применения одних функции к другим.

Основным понятием является понятие функции *f*, которая сопоставляет по крайней мере один объект *f(x1, …, xn)* (ее значение) с каждой *n* – кой объектов *x1, …, xn* (ее аргументов, которые сами могут рассматриваться как функции). λ - исчисление позволяет учитывать специфику вычисления функции в зависимости от используемых аргументов, то есть от того какой из аргументов рассматривается как связанная переменная.

Например, в дифференциальном исчислении выражение *x-y* может рассматриваться как функция *f* от *x*, либо как функция *g* от *y*. Для того, чтобы их различать будем записывать:

*f = λx.[x-y], g = λy.[x-y].*

Говорят, что префикс *λx* абстрагирует функцию *λx.[x-y]* от выражения *x-y.*

Аналогичные обозначения применяются для функций от многих переменных. Например, *x-y* отвечает функциям от двух переменных *h* и *k*: *h(x, y) = x-y, k(y, x) = -y+x.* Это можно записать:

*h = λxy.[x-y], k = λyx.[x-y].*

Однако, можно избежать использования функций от нескольких переменных, если использовать функции, значениями которых являются функции.

Например, определим функцию:

*s = λx.[λy.[x-y]],*

которая для каждого числа a превращается в

*s(a) = λx.[λy.[x-y]](a) = λy.[a-y],*

а для каждой пары *a, b* в

*s(a (b)) = s(a, b) = λx.[λy.[x-y]](a, b) = a-b.*

Предположим, что имеется бесконечная последовательность переменных и конечная или бесконечная последовательность констант. Атом определяется как переменная или константа. Множество λ - термов определяется индуктивно:

1. Каждый атом есть λ - терм.
2. Если *X* и *Y* - λ - термы, то (*XY)* - λ - терм.
3. Если *Y* - λ - терм, а *x –* переменная, то *λx.[Y]* - λ - терм.

Приведенное выше определение функции *g(x)* в этом исчислении записывается следующим образом:

*g = λ(x).[a\* sin(p \* x + q)],*

а вместо *g(t2 + 2)* можно записать:

*λ(x).[a\* sin(p \* x + q)] (t2 + 2).*

За символом λ следует еще несколько синтаксических конструкций: вначале идет список связанных переменных, затем выражение, в которое входят эти переменные (тело λ - выражения). Если остановиться здесь, то будем иметь функцию без операндов, такую как sin (заголовок функции). Но далее может следовать список фактических операндов. В этом случае имеем результат: взятие функции от этих операндов.

Важное значение приобретает сочетание λ - исчисления и рекурсии. Предположим, что существует функция «следующий» - назовем ее *next(x)* – для каждого целого положительного числа и нуля. На самом деле – это функция *x + 1*, но будем считать, что + пока не определен.

Используя *next*, можем задать функцию «предыдущий» - *prior(x)* (после определения – эта функция будет иметь вид *x –1*). Определим эту функцию следующим образом:

*prior(x) = previous(x, 0)* Where *previous(y, z) = iif(next(z) = y, z, previous(y, next(z))).*

Процесс вычисления *prior(x)* начинается при *z = 0*, далее последовательно вычисляются последовательно функции *next* до тех пор, пока следующий для *z* не будет равен *x.* Это значение *z* и является ответом. Теперь можем определить сумму и разность двух чисел.

*Sum(x, y) = iif(y = 0, x, Sum(next(x), prior(y))),*

*Diff(x, y) = iif(y = 0, x, Diff(prior(x), prior(y))).*

Обратим внимание, что если *y>x*, то при вычислении *Diff* настанет такой момент, когда потребуется вычисление *prior(0)*, а среди положительных чисел нет предшественника нуля. Поэтому говорят, что *Diff* вычислимо только частично для положительных чисел.

Теперь представим *Sum* в λ - исчислении:

Sum = λ(x, y).[iif(y = 0, x, Sum(next(x), prior(y)))]

Можно выполнить дальнейшее преобразование этой функции с помощью λ - исчисления. Введя еще одно λ - обозначение, убираются все рекурсивные ссылки из тела λ - определения:

*Sum =λf*.*λ(x, y).[iif(y = 0, x, Sum(next(x), prior(y)))](Sum),*

а затем можно совсем убрать объект определения из правой части за счет введения оператора Y такого, что если *F* – функция, то Y*F* – решение уравнения *y = F(x)*.

*Sum =* Y*λf*.*λ(x, y).[iif(y = 0, x, f(next(x), prior(y)))].*

В этом определении *f* – связанная переменная, которая при разрешении уравнения может быть заменена на что-либо другое. Следовательно, при замене на *Sum* получим:

*Sum =* Y*λSum*.*λ(x, y).[iif(y = 0, x,.Sum(next(x), prior(y)))].*

В результате введенных обозначений функции принимают форму оператора присваивания (= является приказом), и, как следствие, понятия переменной, константы, литерала могут применяться к функциям также, как и к другим видам значений.

Говорят, что λ - исчисление используется для того, чтобы выполнить операцию λ - редукции и упростить выражение.

Учитывая, что выражение *λx.[Px](a)* может быть редуцировано к *Pa*, выражение:

*λf.λy.λx.[f(x, y)],*

сформулированное:

*λf.λy.λx.[f(x, y)](подсистема)*

сводится к

*λy.λx.[подсистема(x, y)],*

*λy.λx.[подсистема(x, y)](ЭВМ)→λx.[подсистема(x, ЭВМ)],* а

*λx.[подсистема(x, ЭВМ)](процессор)→подсистема(процессор, ЭВМ).*

###### Таким образом, λ - редукция осуществляется слева направо, и поэтому λf.λy.λx.[f(x, y)], сформулированное в виде λf.λy.λx.[f(x, y)](любит, Мария, Иван), дает любит(Иван, Мария).

*3.3.1 Практические задания*

1.Определить функцию *f(x, y) = iif(x>y, sin(x), cos(x))* в λ - исчислении. Дать ее запись для *x=t* и *y=π/2*. Осуществить редукцию.

2.Осуществить редукцию следующих λ - выражений:

*λf.[λg.[λt.[f(g(x)+t(y)]]](sin, cos, tg),*

*λg.[λt.[λf.[f(g(x)+t(y)]]](sin, cos, tg),*

*λf.[λt.[λg.[f(g(x)+t(y)]]](sin, cos, tg),*

**3.4 Использование списков**

Введем некоторые определения. Символ - это набор букв, цифр и специальных знаков. Кроме символов будем использовать: числа, Т – истина, Nil – ложь или пустой список. Будем понимать под константами – числа, Т, Nil. Будем понимать под атомами символы и числа. Назовем списком упорядоченную последовательность, элементами которой являются атомы или другие списки (подсписки). Будем заключать списки в круглые скобки, а элементы списка разделять пробелами.

Формально список можно определить следующим образом:

**Список :- Nil / (голова ∩ хвост)**

[Список либо пуст, либо это пара голова и хвост]

**голова :- атом / список**

[рекурсия в глубину]

**хвост :- список**

[рекурсия в ширину]

Другой вариант определения:

**Список :- Nil / (элемент элемент … )**

**Элемент :- атом / список**

[рекурсия]

Обратим внимание, что атом это не список, хотя символ может идентифицировать список. Каждый символ имеет значение. Им может быть список, в том числе и пустой, константа, функция, которую рассматривают как специальный символ. Для записи функций будем использовать префиксную нотацию, т.е. x + y будем записывать, как (+ x y).

Атомы и списки будем называть S – выражениями.

Для представления списков будем использовать совокупность ячеек памяти, каждая из которых содержит два указателя. Первый указатель играет роль указателя на сына, второй – на брата. Например, список (А В) будет иметь вид.

|  |  |
| --- | --- |
|  |  |

|  |  |
| --- | --- |
|  | ψ |

А

В

Отсутствие указателя будем обозначать символом ψ.

Список (\* (+ 2 3) с) будет иметь представление:

\*

с

+

2

3

Чтобы определить выражение по его машинному представлению, нужно просматривать представление, следуя указателям, расположенным слева, на наибольшую глубину и затем по правым указателям (внешний просмотр).

При записи выражения каждой стрелке, не заканчивающейся на атоме, соответствует открывающаяся скобка, каждому символу ψ - закрывающаяся скобка, каждому атому – сам атом.

|  |  |
| --- | --- |
|  |  |

|  |  |
| --- | --- |
|  | ψ |

|  |  |
| --- | --- |
|  | ψ |

|  |  |
| --- | --- |
|  | ψ |

(1

(2

2)

А

1)

(3

3)

В

Этой схеме соответствует список ((А)(В)).

Вернемся к определению атома. Фактически атомы являются функциями, аргументы которых представляют собой следующие за ними S – выражения. В свою очередь аргумент сам может быть функцией, которую надо вычислить. Поэтому возникает необходимость определять, что представляет собой данный элемент: значение выражения L или символьное имя L какого-то S – выражения. В первом случае перед выражением ставится‘ или указатель QUOTE. Апостроф запрещает вычисление следующего за ним S - выражения, которое воспроизводится без изменений.

Для задания выражений введем функцию Setq:

(Setq <атом> <S – выражение>)

(Setq <атом> ‘<S – выражение>)

В первом случае выражение должно быть вычислено, во втором – на вычисляется. Кроме того, Setq связывает полученный результат с атомом.

(Setq L1 (+ 4 5))

(Setq L1 ‘(+ 4 5))

В первом случае L1 связано с (9), во втором – с (+ 4 5).

Функция CAR возвращает в качестве значения первый элемент списка, то есть его голову. Функция имеет смысл, только для аргументов, являющихся списками, имеющими голову.

(CAR ‘ (A B C)) → A

(CAR ‘ A) → ошибка, А не список.

Функция CDR возвращает хвост списка. Если список состоит из одного элемента, то результатом является Nil.

(CDR ‘ (A B C)) → (B C)

Комбинация вызовов CAR и CDR выделяет произвольный элемент списка. Для простоты эту комбинацию записывают в виде:

(C…R список)

Вместо многоточия записывается комбинация из букв А (для CAR) и D (для CDR).

(CADDAR L) = (CAR (CDR (CDR (CAR L))))

Функция CONS строит новый список из переданных ему головы и хвоста.

(CONS голова хвост)

(CONS ‘a ‘(b c)) → (a b c)

(CONS ‘(a b) ‘(c d)) → ((a b) c d)

(CONS (+ 1 2) ‘(+ 4)) → (3 + 4)

Здесь первый аргумент без апострофа, поэтому берется как значение.

(CONS ‘(+ 1 2) ‘(+ 4)) → ((+ 1 2) + 4)

(CONS ‘(a b c) Nil) → ((a b c))

(CONS Nil ‘(a b c)) → (Nil a b c)

Предикат ATOM используется для анализа списков, а именно, для идентификации является ли объект атомом или нет.

(ATOM ‘x) → T

(ATOM (CDR ‘(A B C))) → Nil. Атом от списка (B C) естественно False.

(ATOM (ATOM (+ 2 3))) → T.

Результат сложения чисел атом. Т также является атомом.

(EQUAL <S-выр1> <S-выр2>) → T, если и только если значения обоих выражений равны.

(EQ <S-выр1> <S-выр2>) → T, если и только если значения обоих выражений равны и представляют один физический список.

(AND <S-выр1> <S-выр2> … <S-вырn>) – если встречается Nil,, возвращается Nil, иначе значение последнего выражения.

(OR <S-выр1> … <S-вырn>) – если при просмотре встречается выражение отличное от Nil, то оно возвращается, иначе Nil.

(NOT <S-выр>) → T, если и только если значение выражения есть Nil.

Определять функции будем согласно λ - исчислению.

(λ (x1, x2, …, xn) f)

xi – формальный параметр.

f – тело функции.

Например, функция вычисляющая сумму квадратов будет определяться следующим образом:

(λ (x y) (+ (\* x x) (\* y y))).

Чтобы произвести вычисления будем осуществлять λ - вызов.

(λ - выражение a1, …, an),

Здесь ai – форма , задающая фактические параметры.

((λ (x y) (+ (\* x x) (\* y y))) 2 3) - дает 13.

Для того чтобы определить новую функцию будем использовать запись:

(DEFUN <имя> <λ - выражение>).

Для удобства исключим значок λ и внешние скобки.

(DEFUN проценты (часть целое) (\* (/ часть целое) 100)).

Вызов:

(проценты 4 20)

Результат: 20.

Условное выражение COND имеет следующий вид:

(COND (p1 a1)

(p2 a2)

. . .

(pn an))

Значение COND определяется следующим образом:

1. Выражения pi вычисляются последовательно слева направо (сверху вниз) до тех пор, пока не встретится выражение, значение которого отлично от Nil.
2. Вычисляется результирующее выражение ai соответствующее этому pi и полученное значение возвращается в качестве результата.
3. Если нет ни одного pi, значение которого отлично от Nil, то значение COND равно Nil/

В более общем виде:

(COND (p1 a1)

…

(pj)

…

(pk ak1 … akn)

…)

В этом случае:

* + если условию не ставится в соответствие результирующее выражение, то в качестве результата при истинности предиката выдается само значение предиката;
  + если условию соответствует несколько S – выражений, то при его истинности они вычисляются слева направо и результатом будет значение последнего S – выражения.

Реализуем логические связки И, ИЛИ, →.

(defun И(x y) (COND (x y) (T Nil)))

(defun ИЛИ (x y) (COND (x T) (T y)))

(defun → (x y) (COND (x y) (T T)))

С помощью COND можно реализовать различные варианты условных выражений.

(if <условие> <то-выр> <иначе-выр>) ≡ (COND (<условие> <то-выр>) (T <иначе-выр>))

Чтобы говорить о некотором свойстве, связанным с некоторым объектом и его значением, будем использовать функцию:

(PUTPROP <атом1> <список> <атом2>).

Эта функция связывает свойства, выраженные списком <атом2>, с конкретным объектом с именем <атом1>. Конкретные значения свойства задаются в объекте <список>.

(PUTPROP ‘Петр ‘(Иван Ирина) ‘Дети).

Чтобы узнать обладает ли объект данным свойством, будем использовать функцию GET:

(GET <атом1> <атом2>) → значение свойства.

(GET ‘Петр ‘Дети) → (Иван Ирина)

Теперь сформулируем алгоритм для вычисления списков. Алгоритм должен вычислять значение списка в последовательности, задаваемой указателями – «сыновьями», а затем учитывается последний из встреченных указателей «братьев». После этого учитываются указатели «сыновья», связанные с этим последним, и так далее, пока не получается окончательное значение для данной ветви. Затем вычисления повторяются снова, охватывая выражения более высокого уровня, и так до тех пор пока не будет определено значе ние всего списка. Те части всего выражения, которые ожидают вычисления значений их аргументов, называются квазарами.

Этот алгоритм представим в вида процедуры Eval(S), где S – выражение.

Sub Eval(S)

If S есть атом then

Возврат ← значение S

Else

If S1 = QUOTE then

Возврат ← S2

Else

S1 есть функция

If S1 указывает на специальную обработку, выполнить эту обработку

Else

Применить Eval ко всем элементам S2 последовательно и затем рекуррентно использовать их найденные значения.

Окончательный возврат ← S1 (Eval (S2))

End If

End If

End If

S1 – первый сын для S. S2 – элементы выражения S, представляющие собой множество братьев выражения S1.

*3.4.1 Практические задания*

1.Создать машинное представление для списков:

*(a (b c) ( (d) e (f)))*

*( + a (\* b (/ (+ c (/ d e)) f)))*

*(/ (+ ( a (- (b c)))) (\* (/ (d b)) a))*

*(flat (hd (a) flat (tl (a) b)))*

*(cons (copy (hd (l) )) copy (tl (l)))))*

2.Определить значения списков:

*(car ‘(a (b c) ((d) e (f))))*

*(cdr ‘(a (b c) ((d) e (f))))*

*(cdadr ‘(a (b c) ((d) e (f))))*

*(cons nil ‘(суть пустой список))*

*(cons (car ‘(a b))(cdr ‘(a b)))*

*(car ‘(car (a b c)))*

*(cdr (car (cdr ‘(a b c))))*

3.Запишите последовательность вызовов CAR и CDR, выделяющих из приведенных списков символ «цель». Упростите эти последовательности с помощью функций C…R.

*(1 2 цель 3 4)*

*((1) (2 цель ) (3 4 цель)))*

*((1 (2 (3 4 цель))))*

4.Определить вид списка, имеющего следующее машинное представление:

|  |  |
| --- | --- |
|  |  |

|  |  |
| --- | --- |
|  |  |

|  |  |
| --- | --- |
|  | ψ |

|  |  |
| --- | --- |
|  |  |

|  |  |
| --- | --- |
|  | ψ |

|  |  |
| --- | --- |
|  |  |

|  |  |
| --- | --- |
|  | ψ |

|  |  |
| --- | --- |
|  |  |

|  |  |
| --- | --- |
|  | ψ |

Это

список

4. Реализовать с помощью COND логические операции: эквивалентность, штрих Шеффера, стрелка Пирса.

5. Реализовать функцию reverse, переставляющую местами элементы списка.

*3.4.2 Компьютерное задание*

Создать БД для индуктивного вывода и реализовать программу заполнения этой базы для оператора Setq. Возможная структура таблицы для хранения списков:

|  |  |  |
| --- | --- | --- |
| Имя поля | Тип | Комментарий |
| List | Text | Наименование списка |
| BeginList | Boolean | Является ли началом списка, если да -TRUE |
| SonPointer | Long | Указатель на сына |
| BrotherPointer | Long | Указатель на брата |
| ValueEl | Text | Значение. Для указателей Nil |
| ValueType | Text | Тип значения: ячейка, список, пустой список, атом, функция |

Пример заполнения:

Пусть последовательно создаются 2 списка:

Setq a (\* (+ 2 3) c)

Setq Second ((a) (b))

Структура списков: Список а

\*

с

+

2

3

Список Second

a

b

Таблица для этих списков будет иметь вид:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| List | BeginList | SonPointer | Brother  Pointer | ValueEl | ValueType |
| a | true | 2 | 4 | Nil | ячейка |
| a | false | 3 | 0 | Nil | ячейка |
| a | false | 0 | 0 | \* | функция |
| a | false | 5 | 11 | Nil | ячейка |
| a | false | 6 | 7 | Nil | ячейка |
| a | false | 0 | 0 | + | функция |
| a | false | 8 | 9 | Nil | ячейка |
| a | false | 0 | 0 | 2 | атом |
| а | false | 10 | 0 | Nil | ячейка |
| а | false | 0 | 0 | 3 | атом |
| а | false | 12 | 0 | Nil | ячейка |
| a | false | 0 | 0 | c | Пустой список |
| Second | true | 14 | 16 | Nil | ячейка |
| Second | false | 15 | 0 | Nil | ячейка |
| Second | false | 0 | 0 | a | список |
| Second | false | 17 | 0 | Nil | ячейка |
| Second | false | 18 | 0 | Nil | ячейка |
| Second | false | 0 | 0 | b | Пустой список |

Порядок выполнения

1. Создание таблиц и форм для ввода и хранения списков (1 занятие).
2. Реализация алгоритмов преобразования описания списка в машинное представление и обратно (2 занятия).
3. Реализация операций над списками (2 занятия)
4. Реализация алгоритма Eval (2 занятия).
5. Оформление результатов (1 занятие).

# 4. Логическое программирование

Логическое программирование основывается на представлении задач в виде теорем, доказываемых в рамках исчисления предикатов первого порядка.

**4.1 Модели и опровержения**

В доказательстве теорем стараются показать, что определенная ППФ *B* есть логическое следствие множества ППП *S = {A1, …, An}*, называемых аксиомами рассматриваемой задачи. Правило вывод есть правило, при помощи которого из ранее полученных выражений можно получить новое. Например, если *Ai* и *Aj* истинные ППФ, то запись:

**

указывает, что *Ak* можно получить из *Ai* и *Aj* с помощью правила *fq* . Рассмотрим последовательность *S(0), …, S(j+1),* образованную из некоторого множества аксиом *S* по правилу:

*S(0) = S,*

*…*

*S(j+1) = S(j) ∪ F(S(j),*

где *F(S)* – множество выражений, которое можно получить из множества *S*, применяя к каждому его элементу все возможные правила *fq* из конечного множества *F={fq}.* Говорят, что высказывание *B* следует из аксиом, принадлежащих *S,* если *B ∈ S(j)* для некоторого *j.*

Напомним, что терм – это константа, переменная или функция. В качестве своих аргументов n – местная функция должна иметь п термов. Таким образом, термами будут: *a, b, c, f(a), g(f(x), y).*

Атомом называется предикат со своими аргументами. Литерал – это атом или его отрицание. Предложение *C –* это дизъюнкция литералов. Множество предложений *S* интерпретируется как единое высказывание, представляющее конъюнкцию всех его предложений.

Как уже отмечалось, высказывание *T* следует из множества предложений *S*, если *T* есть логическое следствие предложений из *S.* Предположим для простоты, что *T* состоит из единственного предложения. Рассмотрим множество предложений:



Для того, чтобы высказывания *S∪T* были истинными, все предложения *Ci* и *T* должны быть истинными. В свою очередь значения истинности этих предложений будут определяться значениями истинности, содержащихся в них атомов, причем значения истинности должны присваиваться атомам так, чтобы, по крайней мере, один литерал в каждом предложении был истинным.

Отдельное присваивание истинности атомам называется моделью. Если *S* влечет за собой *T,* то не существует модели, в которой *S* истинное, а *T* – нет. Вместе с тем, если высказывание *T* истинное, то его отрицание должно быть ложным. Поэтому, если *S* влечет за собой *T*, высказывание:

 (1)

должно быть ложным для любой модели.

Этот же вывод можно сделать следующим образом:



и от противного:



Предположим, что число атомов конечно. Тогда существует конечное множество моделей, так как *k* атомам можно присвоить логические значения *2k* различными способами. Очевидно, что присваивать значения этим комбинациям можно последовательно и если для всех моделей (1) окажется ложным, то, безусловно, *S* влечет *T.* Такое доказательство называется методом от противного и составляет основу методов доказательства теорем.

Доказать противоречивость (1), если число атомов конечно, достаточно просто. Просто ли это на практике зависит от количества атомов и от возможности порождать и проверять модели.

Возникает задача выявления условий, гарантирующих конечное число атомов. Пусть *S* – это множество высказываний истинных для тех атомов, которые непосредственно входят в *S* и тех, которые из них можно вывести. Последнее множество может быть бесконечным. В этом можно убедиться на следующем примере. Пусть *S* – высказывание, содержащее единственный атом *S = {R(a, x)}.*

Предположим, что определена функция *g(x).* Тогда можем построить бесконечную последовательность *R(a, x), R(a, g(x)), R(a, g(g(x))), …* . Эта последовательность перечислимая, так как можно придумать схему перечисления, по которой упорядочиваются высказывания. Например, по уровню вложенности второго аргумента. Можно показать, что всегда можно построить схему перечисления бесконечного множества атомов, полученного из конечного при помощи некоторой подстановки.

Предположим, что *S* не содержит переменных. Такая ситуация называется основным случаем, а соответствующий универсум (область определения) Эрбрановой базой. В основном случае можно провести перечисление.

Докажем, например, для конкретной пары чисел *(a, b)* (но не для произвольной пары чисел вообще *x, y*) истинность высказывания:

*a = b → a ~~>~~ b.*

Аксиомами будут:

*A1: a > b,*

*A2: a < b,*

*A3: a = b.*

Соответствующие предложения будут иметь вид:

 - выполняется одно из отношений;

 - одновременно не выполняются никакие два отношения.

В этих обозначениях наше утверждение имеет вид:



и его отрицание *A3A1.* Отсюда получим:



Обозначим 

Так как в *S* может быть только 23 = 8 атомарных высказываний, легко построить все возможные модели. Если для каждой из них хотя бы одно предложение в *S* принимает значение ложь, то высказывание вида (1) будет ложным, и поэтому наше утверждение будет истинным.

|  |  |
| --- | --- |
| Модель | Предложения, которым присваивается значение ложь |
| *A1, A2, A3* | *C2, C3, C4* |
|  |  |
|  | *C3* |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

# Этот метод неэффективен и избыточен. Можно показать, что высказывание *S* противоречиво, если исследовать меньшее число моделей. Достаточно ограничиться атомами *A1* и *A3*, то есть провести присваивание значений истинности только этим атомам, а значит использовать только четыре модели.

Теорема Эрбрана – Сколема – Геделя. В этой теореме утверждается, что можно найти частичное присваивание, приписывающее значение ложь любому противоречивому множеству предложений.

**4.2 Доказательство от противного**

*4.2.1 Неаксиоматическое описание процедуры*

В логике предикатов используется формальный метод доказательства теорем, допускающий возможность его машинной реализации. Но для упрощения рассмотрим сначала процедуру доказательства неаксиоматическим путем.

Пусть в качестве *∑* задана группа логических формул следующего вида:

*Отец(x, y)* : *x* является отцом *y*;

*Брат(x, y) : x, y –* братья;

*Кузен(x, y) : x, y –* двоюродные братья;

*Дедушка(x, y) : x –* дедушка y.

Используя эти предикаты можно записать следующие утверждения:

*(1) (∀x1)(∀y1)(∀z1)((отец(x1, y1) ∧ отец(y1, z1)) → дедушка(x1, z1)):*

если *x1* отец *y1,* а *y1* отец *z1*, то *x1* дедушка *z1.*

*(2) (∀x2, y2, z2, w2)((отец(x2, y2) ∧ брат(x2, z2) ∧ отец(z2, w2)) → кузен(y2, w2)) .*

*(3) (∀x3, y3, z3)((отец(x3, y3) ∧ отец(x3, z3) ∧ ) → брат(y3, z3)).*

Предположим также, что помимо этих логических формул заданы следующие факты:

*(4) Отец(Александр, Сергей),*

*(5) Отец(Сергей, Ричард).*

*(6) Отец(Сергей, Иван).*

Сначала неаксиоматически зададим процедуру, которая из логических формул выводит заключение. Вывод формулы *C* из логических формул *A, B* будем задавать в виде схемы:

*A:(∀x)(человек(x)→смертен(x))*

*B:человек(Сократ)*

***x/Сократ***

*C: смертен(Сократ)*

А: Все люди смертны

В: Сократ – человек

С: Сократ – смертен.

Если в переменную подставляется значение, то слева от наклонной черты запишем имя переменной, а справа – значение. Например, если в переменную *x* подставляется значение *Сократ*, то будем это записывать как *x/Сократ.*

Предположим, теперь, что требуется доказать – истинна или ложна логическая формула. В ее истинности или ложности можно убедиться логической процедурой, применив ранее заданные логические формулы. Эта процедура выглядит аналогично процедуре доказательства теорем, когда выдвигается некоторая теорема, а справедливость ее устанавливается из известных аксиом.

Например, предположим, что задан вопрос: «Кто дедушка Ричарда?». Его можно записать в виде:

*(∃x)(дедушка(x, Ричарда)).*

Ответ может быть получен в следующей последовательности:

(1)

(4)

*x1/Александр*

*y1/Сергей*

*(∃z1)(отец(Сергей, z1)→дедушка(Александр, z1))*

(5)

*z1/Ричард*

*Дедушка(Александр, Ричард)*

Поскольку заключение получается из комбинации уже существующих правил, ответ является результатом процедуры восходящего вывода.

*(∃x)(дедушка(x, Ричард))?*

(1)

*x/x1*

*z1/Ричард*

*(∃x1)(∃y1)(отец(x1, y1)∧отец(y1, Ричард))?*

(5)

*x1/Сергей*

*(∃x1)(отец(x1, Сергей))?*

(4)

*x1/Александр*

истина

С другой стороны можно осуществить и нисходящий вывод. Нисходящий вывод начинается с того заключения, которое должно быть получено, и далее в базе проводится поиск информации, которая последовательно позволяет прийти к данному заключению.

В логике предикатов такая процедура полностью формализована и носит название метода резолюции.

Для применения этого метода исходную группу логических формул преобразуют в нормальную форму. Преобразование проводится в несколько стадий.

*4.2.2 Процесс нормализации*

*4.2.2.1Префиксная нормальная форма*

Рассмотрим фразу «любой человек имеет отца». Ее можно представить в логике предикатов в следующем виде:

*(∀x)(человек(x) → (∃y)(отец(y, x))).*

∃ находится внутри ППФ. Это не всегда удобно, поэтому целесообразно вынести все кванторы за скобки, чтобы они стояли перед ППФ. Такая форма носит название префиксной нормальной формы.

Преобразование к префиксной нормальной форме произвольной ППФ можно легко выполнить машинным способом, используя для этого несколько простых правил.

Во-первых, необходимо исключить связки → и ≈. По определению:



Следовательно, во всех комбинациях ППФ можно ограничиться тремя связками ∨, ∧ и ‾ .

Предикат, не содержащий переменных, аналогичен высказыванию, поэтому такие предикаты будем обозначать *F, G.* Когда некоторое выражение выполняется для любого квантора ∀ и ∃ будем записывать такой квантор *Q*.

Легко показать:

1.

Это соответствует: высказывание не имеет отношения к квантору переменной, которая включена в другой предикат.

Между предикатами существует следующее соотношение:



Таким образом, «необязательно *F(x)* выполняется для всех *x*» ≡ «существует такое *x*, что *F(x)* не выполняется»; «не существует такое *x*, что выполняется *F(x)*» ≡ «для всех *x* не выполняется *F(x)*».

Далее имеем:

2.*(∀x)F(x)∧(∀x)H(x) = (∀x)(F(x)∧H(x))*

*(∃x)F(x)∨(∃x)H(x) = (∃x)(F(x)∨H(x)).*

Это означает, что квантор ∀ обладает дистрибутивностью относительно ∧, а ∃ - относительно ∨.

С другой стороны:

*(∀x)F(x)∨(∀x)H(x)≠(∀x)(F(x)∨H(x)).*

*(∃x)F(x)∧(∃x)H(x)≠(∃x)(F(x)∧H(x)).*

Это легко показать. Пусть для первого выражения *x* определен на множестве *D ={а, б, в}*, при этом истиной являются *F(а), F(б), H(в).* Тогда левая часть не выполняется, а правая истинна. На самом деле справедливо:

*(∀x)F(x)∨(∀x)H(x)→(∀x)(F(x)∨H(x)).*

*(∃x)F(x)∧(∃x)H(x)→(∃x)(F(x)∧H(x)).*

Смысл этих формул в том, что описание для двух разделенных предикатов не совпадает с описанием, использующим эти предикаты одновременно как одно целое и с одинаковыми переменными. Поэтому, изменив все переменные в их правых частях, получим:

*(∀x)F(x)∨(∀x)H(x) = (∀x)F(x) ∨ (∀y)H(y),*

*(∃x)F(x) ∧(∃x)H(x) = (∃x)F(x) ∧ (∃y)H(y).*

Теперь *H(y)* не содержит переменной *x* и поэтому не зависит от связывания *x.* Отсюда:

*3. (∀x)F(x)∨(∀x)H(x) = (∀x)(∀y)(F(x) ∨ H(y)),*

*(∃x)F(x) ∧(∃x)H(x) = (∃x)(∃y)(F(x) ∧ H(y)).*

Таким образом, процесс получения префиксной нормальной формы можно представить как последовательность шагов:

1. Используя формулы для →, ≈, заменим их на ∧, ∨, ⎯.
2. Воспользовавшись выражениями



внесем отрицания внутрь предикатов.

1. Вводятся новые переменные, где это необходимо.
2. Используя 1, 2, 3 получаем префиксную нормальную форму.

В качестве примера рассмотрим следующее выражение:



Шаг1: 

Шаг2: 

Шаг 3: нет необходимости.

Шаг 4: используем выражение 1 по переменной *z* –



Теперь применяем выражение 1 по переменной *w –*



*4.2.2.2 Скалемовская нормальная форма*

Дальнейшее преобразование предполагает полное исключение из формулы кванторов. При этом необходимо сохранить семантику. Это достигается за счет введения скалемовских функций. Для описания этой функции рассмотрим следующий пример: *(∀x)(∃y)подсистема(x,y),* для каждого *x* существует *y* такой, что *x* его подсистема.

Это означает, что если выделить конкретное *x*, то для этого *x* существует *y*, удовлетворяющее функции *подсистема(x, y).* Иными словами, *y* зависит от *x,* то есть, если задано *x*, то и существует соответствующее ему *y*. Отсюда следует, что *y* можно заменить на функцию *f(x),* которая задает отношение между *x* и *y.* Поскольку *f(x)* возникло из-за того, что переменная *y* квантифицирована ∃, при подстановке функции на место *y*, квантор ∃ уже не требуется. Таким образом, исходную логическую формулу можно переписать: *(∀x) подсистема(x, f(x).*

Такая функция называется скалемовской.

Приоритетность действия кванторов, имеющихся в префиксной форме, определяется слева направо. Например: *(∀x)(∃y)(∀z)F(x, y, z) ⇒ (∀x)(∀z)F(x, f(x), z).* А для случая: *(∀x)(∀z)(∃y)F(x, y, z) ⇒ (∀x)(∀z)F(x, f(x, y), z).*

Таким образом, переменной, которая в логической формуле влияет на связанную квантором существования переменную, является любая переменная с квантором общности, которая стоит левее переменной из квантора существования. Если переменная связанная квантором существования является крайней слева, такую формулу можно заменить функцией без аргументов (константой):

*(∃x)(∀y)подсистема(x, y) =подсистема.*

Если проделать эту операцию для примера, получим:



В случае, когда в одной логической формуле имеется более двух связанных квантором существования переменных, то сколемовская функция также будет распадаться на различные функции.

*(∀x)(∃y)(∀z)(∃w)F(x ,y ,z, w) = (∀x)(∀z)F(x, f(x), z, g(x, z)).*

Если подобным образом исключить связанные квантором существования переменные, то любые другие переменные будут связаны только квантором общности и не надо пояснять, что преременные связаны этим квантором. Следовательно, квантор общности можно исключить. Для примера получим:



Такое представление и есть сколемовская нормальная форма.

Теперь можем определить алгоритм получения сколемовской формы:

1. Составить список *L* переменных, связанных квантором общности. Сначала список *L* пуст.
2. Пусть *Ci* – рассматриваемое предложение, *j* – номер использующейся сколемовской функции *Sij.* Положим *j = 1.*
3. Удалить кванторы , начиная с самого левого и применяя в зависимости от необходимости правила а или б.

а. Если рассматривается квантор *∀x,* то удалить его и добавить *x* к списку *L.*

б. Если рассматривается квантор *∃x,* то удалить его и заменить *x* во всем предложении *Ci* термом *Sij(x1, …, xk),* где *x1, …, xk* переменные, находящиеся в этот момент в списке *L*. Увеличить *j* на 1.

Следующим шагом является переход к конъюнктивной нормальной форме, а затем к клаузальной форме, то есть форме предложений.

Приведение к конъюнктивной нормальной форме осуществляется заменой пока это возможно *(A∧B)∨C* на *(A∨C)∧(B∨C).* В результате получим выражения вида *A1∧ … ∧An,* где *Ai* имеет вид (, причем  есть терм или атомарный предикат или атомарная формула или их отрицание. Целесообразно осуществить и минимизацию по следующим правилам:

**

Наконец, исключаем связку ∧, заменяя *A∧B* на две формулы *A, B.* В результате многократной замены получим множество формул, каждая из которых является предложением. Это и есть клаузальная нормальная форма.

*4.2.2.3 Предложения Хорна*

Все клаузальные формулы (предложения) представляют собой несколько предикатов, то есть:



Эти формулы в зависимости от *k, l* классифицируются по нескольким типам.

* 1. Тип 1: *k = 0, l = 1.*

Предложение представляет собой единичный предикат, и оно может быть записано как *→F(t1, t2, …, tm),* где *t1, t2, …, tm* – термы. В случае когда все термы представляют собой константы, они описывают факты, которые соответствуют данным из БД. Если термы содержат переменные, то они задают общезначимые представления, высказываемые для группы фактов. Например, таким представлением является:

*→ текучесть(x)*: все течет, все изменяется.

* 1. Тип 2: *k≥1, l = 0.*

Этот тип можно записать в виде:

*F1∧F2∧…∧Fk→.*

Обычно используется для описания вопроса. Причина, по которой данный тип предложения представляется вопросом, является то, что ответ на вопрос реализуется в виде процедуры доказательства, однако для этого доказательства используется метод опровержения, при котором создается отрицание вопроса и доказывается, что отрицание не выполняется.

* 1. Тип : *k≥1, l=1.*

Этот тип соответствует записи в форме «Если \_\_\_, то \_\_\_\_.».

*F1∧F2∧…∧Fk→G1.*

* 1. Тип: *k=0, l>1.*

Этот тип имеет вид:

→*G1∨G2∨…∨Gl*

и используется при представлении информации, содержащей нечеткость. Обычно нечеткость возникает при неполноте информации. В этой формуле появляется неполнота информации в том смысле, что из нее нельзя определить, какой из предикатов *G1, …, Gl* является истиной.

* 1. Тип: *k≥1, l>1.*

Это наиболее общий тип.

В системе предложений Хорна среди всех перечисленных типов предложений допустимы типы предложений 1, 2, 3, а 4, 5 запрещены.

*4.2.3Формализация процесса доказательств*

Доказательство демонстрирует, что некоторая ППФ является теоремой на заданном множестве аксиом Σ (то есть результатом логического вывода из аксиом).

Положим, что Σ есть множество из *n* ППФ, а именно, *A1, A2, …, An*, и пусть ППФ, для которой требуется вычислить является ли она теоремой, есть *B*. В таких случаях можно сказать, что доказательство показывает истинность ППФ вида:

(*A1∧…∧An→B) (1)*

при любой его интерпретации. Можно сказать по другому: это определение эквивалентно тому, что отрицание формулы:

 (2)

не выполняется (ложно) при любой интерпретации.

Поскольку эта формула является ППФ, то ее можно преобразовать к клаузальной форме, используя для этого приведенный выше алгоритм. Пусть преобразованная ППФ есть *S.*

Основная идея метода, называемого методом резолюции, состоит:

1. В проверке содержит или не содержит *S* пустое предложение.
2. В проверке выводится или не выводится пустое предложение из *S*, если пустое предложение в *S* отсутствует.

Любое предложение *Ci,* из которого образуется *S*, является совокупностью атомарных предикатов или их отрицаний (предикат и его отрицание называются литералами), соединенных символами дизъюнкции, вида:



Сама же *S* является конъюнктивной формой, то есть имеет вид:



Следовательно, условием истинности *S* является условие истинности всех *Ci* в совокупности.

Условием ложности *S* является ложность по крайней мере одного *Ci.* Однако, условием, что *Ci* будет ложным в какой-нибудь интерпретации, является то, что множество  будет пустым. Это легко показать. Положим, что это не так, тогда среди всех возможных интерпретаций имеется такая, что какой-нибудь из литералов этого множества или все они будут истиной и тогда *Ci* не будет ложью. Следовательно, если *S* содержит пустые предложения, формула (2) является ложью, а это показывает, что *B* выводится из группы предикатов *A1, … , An*, то есть из Σ.

*4.2.3.1 Метод резолюций для логики высказываний*

Предикат, который не содержит переменных, совпадает с высказыванием. Для упрощения рассмотрим сначала резолюцию для высказываний.

Предположим, что в множестве предложений есть дополнительные литералы (которые отличаются только символами отрицания *L* и **) вида:



Исключим из этих двух предложений дополнительные литералы и представим оставшуюся часть в дизъюнктивной форме (такое представление называется резольвентой):



После проведения этой операции легко видеть, что *C* является логическим заключением *Ci* и *Cj.* Следовательно, добавление *C* к множеству *S* не влияет на вывод об истинности или ложности *S*. Если выполняется *Ci = L, * то *C* пусто. То что *C* является логическим заключением из *S* и *C* пусто, указывает на ложность исходной логической формулы.

Приведем простой пример такого доказательства:



Получим доказательство принципом резолюции:



 - пустое предложение.

Такой вывод теорем из аксиоматической системы носит название дедукции. Алгоритм дедуктивного вывода удобно представлять с помощью древовидной структуры:



P







Q



# Такая структура называется дедуктивным деревом или деревом вывода.

*4.2.3.2 Принцип резолюции для логики предикатов*

Поскольку в логике предикатов внутри предикатов содержатся переменные, то алгоритм доказательства несколько изменяется. В этом случае, перед тем как применить описанный алгоритм, будет проведена некоторая подстановка в переменные и вводится понятие унификации с помощью этой подстановки. Унификацию проиллюстрируем следующим простым примером.

Рассмотрим два предиката – *L(x)* и *L(a).* Предположим, что *x* – переменная, *a* - константа. В этих предикатах предикатные символы одинаковы, чего нельзя сказать о самих предикатах. Тем не менее подстановкой *a* в *x* одинаковыми (эта подстановка и называется унификацией).

Целью унификации является обеспечение возможности применения алгоритма доказательства для предикатов. Например, предположим имеем:



В данном случае *L(x)* и  не находятся в дополнительном отношении. При подстановке *a* вместо *x* будут получены, соответственно,  и , и поскольку эти предикаты отличаются только символами отрицания, то они находятся в дополнительном отношении. Однако, операцию подстановки нельзя проводить при отсутствии каких-либо ограничений.

Подстановку *t* в *x* принято записывать как *{t/x}.* Поскольку в одной ППФ может находиться более одной переменной, можно оказаться необходимо провести более одной подстановки. Обычно эти подстановки записываются в виде упорядоченных пар *{t1/x1, …, tn/xn}.*

Условия, допускающие подстановку:

* + *xi –* является переменной,
  + *ti* – терм (константа, переменная, символ, функция) отличный от *xi,*
  + для любой пары элементов из группы подстановок, например (*ti/xi* и *tj/xj)* в правых чачтях символов / не содержится одинаковых переменных.

Унификация

Обозначим группу подстановок *{t1/x1, …, tn/xn}* через *θ* . Когда для некоторого представления *L* применяется подстановка содержащихся в ней переменных *{x1, …, xn},* то результат подстановки, при которой переменные заменяются соответствующими им термами *t1, …, tn* принято обозначать *Lθ.*

Если имеется группа различных выражений на основе предиката *L,* то есть *L1, …, Lm}*, то подстановка *θ,* такая, что в результате все эти выражения становятся одинаковыми, то есть *L1θ = L2θ = … = Lmθ, θ -* называется унификатором *{L1, …, Lm}.* Если подобная подстановка *θ* существует, то говорят, что множество *{L1, …, Lm}* унифицируемо.

Множества *{L(x), L(a)}* унифицируемо, при этом унификатором является подстановка *(a/x).*

Для одной группы выражений унификатор не обязательно единственный. Для группы выражений *{L(x, y), L(z, f(x)}* подстановка *θ = {x/z, f(x)/y}* является унификатором, но является также унификатором и подстановка *θ = {a/x, a/z, f(a)/y}.* Здесь *a* – константа, *x –* переменная. В таких случаях возникает проблема, какую подстановку лучше выбирать в качестве унификатора.

Операцию подстановки можно провести не за один раз, а разделив ее на несколько этапов. Ее можно разделить по группам переменных, проведя, например, подстановку *{t1/x1, t2/x2, t3/x3, t4/x4}* сначала для *{t1/x1, t2/x2},* а затем для *{t3/x3, t4/x4}*. Допустимо также подстановку вида *a/x* разбить на две подстановки *u/x* и *a/u.* Результат последовательного выполнения двух подстановок *θ* и *λ* также подстановка и обозначается *λ°θ.*

Если существует несколько унификаторов, то среди них непременно найдется такая подстановка *σ,* что все другие унификаторы являются подстановками, выражаемыми в виде *σ°λ*, как сложная форма, включающая эту подстановку. В результате подстановки переменные будут замещаться константами и описательная мощность ППФ будет ограничена.

Чтобы унифицировать два различных выражения предиката, необходима такая подстановка, при которой выражение с большей описательной мощностью согласуется с выражением с меньшей описательной мощностью. Такую подстановку принято называть «наиболее общим унификатором» (НОУ). Метод отыскания НОУ из заданной группы предикатов выражений называется алгоритмом унификации.

Этот алгоритм состоит в том, что сначала упорядочиваются выражения, которые подлежат унификации. Когда каждое выражение будет упорядочено в алфавитном порядке, среди них отыскивается такое, в котором соответствующие термы не совпадают межде собой.

Положим, что при просмотре последовательно всех выражений в порядке слева направо несовпадающими термами оказались *x, t.* Например, получено *{L(a, t, f(z)), L(a, x, z)}*. В этом случае, если:

1. *x* является переменной;
2. *x* не содержится в *t,* к группе подстановок добавляется *{t/x}.*

Если повторением этих операций будет обеспечено совпадение всех изначально заданных выражений, то они унифицируемы, а группа полученных подстановок является НОУ.

В приведенном примере третий терм в одном случае *z,* а в другом – *f(z),* первое условие выполняется, а второе – нарушается. Поэтому подстановка недопустима. Если в группе предикатных выражений остается хотя бы одно такое, для которого никакими подстановками нельзя получить совпадения с другими выражениями, такая группа называется неунифицируемой.

Рассмотрим другой пример:

*P1 = L(a, x, f(g(y))),*

*P2 = L(z, f(z), f(u)).*

1. Первые несовпадающие члены: *{a, z}.*

Подстановка: *a/z.* Имеем:

*P1 = L(a, x, f(g(y))),*

*P2 = L(a, f(a), f(u)).*

1. Первые несовпадающие элементы *{x, f(a)}.* Подстановка: *[f(a)/x].* Имеем:

*P1 = L(a, f(a), f(g(y))),*

*P2 = L(a, f(a), f(u)).*

1. Первые несовпадающие элементы *{g(y), u}.* Подстановка: *[g(y)/u].* Получаем совпадение. Следовательно, НОУ: *[a/z, f(a)/x, g(y)/u].*

##### Алгоритм доказательства

Пусть заданы:



Предикаты  делаются дополнительными с помощью подстановки *[a/x]*. Суждение о том, становятся ли два выражения дополнительными, выносится:

1. По различию используемых символов.
2. По существованию НОУ для двух выражений, в которых удалены одинаковые символы.

Далее все делается рекуррентно.

Пример 1. Милиция разыскивает всех въехавших в страну, за исключением дипломатов. Шпион въехал в страну. Однако, распознать шпиона может только шпион. Дипломат не является шпионом.

Оценим вывод: среди милиционеров есть шпион.

Воспользуемся следующими предикатами:

*Въехал(x): x* въехал в страну.

*Дипломат(x): x* – дипломат.

*Поиск(x, y): x* разыскивает *y.*

*Милиционер(x): x* – милиционер.

*Шпион(x): x* – шпион.

Если выразим через эти предикаты посылку и вывод в форме ППФ, то получим:



для всех *x,* если *x* не является дипломатом, но въехал в страну, найдется милиционер *y,* который его разыскивает.



Если существует шпион *x*, который въехал в страну, и некоторый *y* разыскивает его, то он сам шпион.



Для всех *x* справедливо, что если *x* является шпионом, то он не дипломат.

Заключение:



Существует *x* такой, что он является шпионом и милиционером.

Если эти формулы преобразовать в клаузальную нормальную форму, то получим:



Заключение преобразуем в свое отрицание:



и затем в клаузальную форму без квантора общности.

Последующий процесс доказательства имеет вид:



*дипломат(а)∨милиционер(f(а)) [a/x] из 2,4 (9)*

*милиционер(f(а)) [a/x] из 8,9 (10)*

*дипломат(а)∨поиск(f(а),а) [a/x] из 1,4 (11)*

*поиск(f(а),а) [a/x] из 8,11 (12)*

*шпион(f(a)) [a/x] из 12,5 (13)*

**

* [f(a)/x)] из 10 и 14 (15)*

Еще одной возможностью метода резолюции является возможность получать конкретные значения переменных, содержащихся в заключении.

*4.2.3.3 Задачи, использующие равенства*

Новые предложения получались до сих пор двумя способами: подстановкой и резолюцией. При резолюции пары предложений, отображаются в новые предложения, а подстановка заменяет терм в предложении другим термом той же синтаксической формы. Иногда возникает необходимость заменить терм равным ему термом, который не является термом, для которого возможна подстановка (подстановочным случаем) в первом терме. Рассмотрим простой пример. Положим *f(x, y) = x + y.* При сравнении предложений:



у нас нет способа обнаружить противоречие, поскольку резолюция не допускает замены термов разных синтаксических форм. Поэтому имеет смысл, включить в программу, использующую принцип резолюции, специальные правило вывода, которое должно применяться, когда возникает необходимость в операции равенства. Чтобы это правило было совместимо с остальной частью программы, оно должно удовлетворять двум условиям:

1. Работать с предложениями, в которых равенство выражено в виде атомов.
2. Быть операцией, превращающей два предложения в третье.

Это специальное правило вывода называется парамодуляцией.

Пусть *A, B* и т.д. будут множествами литералов, а *α, β, γ* - термами, то есть константами, переменными или функциональными выражениями. В дополнении к обычному определению атомов и литералов будем записывать атомы равенства в виде *α=β* (терм *α* равен терму *β).* К таким термам можно применять подстановку.

##### Правило парамодуляции

Если для заданных предложений *C1* и *C2 = (α’ = β’, B) (*или *C2’ = (β’ = α’, B))*, не имеющих общих переменных в общей части *B* выполняются условия:

1. *C1* содержит терм *δ.*
2. У *δ* и *α’* есть наиболее общий унификатор:

,

где *ui* и *wj* – переменные соответственно из *α’* и *δ,*

то надо построить предложения =*C1π1,* а затем *C#,* заменяя *α’* на *β’π2* для какого-то одного вхождения *α’* в *C1\**. Наконец вывести:

C3=(C#, Bπ).

Формулировка весьма сложна, но ее реализация очень проста. В простейшем случае *B* пусто, так что предложения, содержащие высказывания с равенством, состоят из единственного атома выражающего равенство. Таким образом, из:

*C1={Q(a)},*

*C2={a=b}*

можно вывести:

*C3={Q(b)}.*

Несколько более сложный случай:

*C1={Q(x)},*

*C2={(a=b)}.*

Подстановка *π = (a/x)* дает:

*C1\*={Q(a)},*

*C3={Q(b)}.*

При одном применении парамодуляции подстановка *α=βπ2* применяется в *С1\** только один раз. Таким образом, если заданы предложения:

*C1={Q(x), P(x)},*

*C2={(a=b)},*

то одно применение парамодуляции с подстановкой *π = (a/x)* дает сначала

*C1\*={Q(a), P(a)},*

а затем или

*C3={Q(a),P(b)},*

либо

*C3={Q(b), P(a)}.*

Для получения *C4={Q(b), P(b)}* требуется повторное применение парамодуляции.

Рассмотрим пример по сюжету известной сказки Андерсона «Принцесса на горошине», который может служить тестом на наличие королевской крови.

Определения для примера:

1. *x, y, z* – переменные, принимающие значения на множестве людей.
2. *M(x)*: *x* – мужчина.
3. *C(x)*: *x –* простолюдин.
4. *D(x)*: *x* может почувствовать горошину через перину.
5. *x = k*: *x* – король.
6. *x = q*: *x* – королева.
7. *d(x,y)*: дочь *x* и *y*.
8. *x = p*: *x* – принцесса.

Исходные предложения:

** - существует мужчина.

 - существует женщина.

 - любой мужчина не простолюдин король.

 - любая королева – женщина и не простолюдинка.

 - дочь короля и королевы – принцесса.

 - принцесса может почувствовать горошину.

Удалим квантор существования из предложений *C1, C2,* обозначив через *f1, f2* сколемовские функции без переменных, так как перед квантором существования нет квантора общности.

 : *f1 –* мужчина.

 *f2 –* женщина.

Опускаем кванторы всеобщности в *C3, C4.* Проводим резолюцию *C1* с *C3,* а затем *C2* и *C4.* Получаем:

 *f1 –* король или простолюдин.

 *f2 –* королева или простолюдинка.

Затем осуществляем парамодуляцию *C7* и *C5*. Это дает:

 дочь королевы и *f1* – есть принцесса или *f1* - простолюдин. Затем осуществляем парамодуляцию *C8* и *C9*. Это дает:

 дочь *f1* и *f2* есть принцесса, либо *f1,* либо *f2* – простолюдин. Наконец парамодуляция *C10* с *C6* дает:

 дочь *f1* и *f2* может почувствовать горошину, либо *f1,* либо *f2 –* простолюдин.

*4.2.4 Стратегии очищения*

Применение принципа резолюции без ограничений может привести к слишком большому количеству предложений, чтобы с ним можно было бы работать на практике. Поэтому надо уметь заранее определять, какие выполнять резолюции и какие производить выводы. На этом и основываются стратегии очищения.

*4.2.4.1 Стратегия предпочтения одночленов*

В стратегии предпочтения одночленов требуется, чтобы на каждом шаге вывода все резолюции, использующие предложения только с одним литералом, вычислялись прежде других резолюций. В общем случае резолюции, использующие короткие предложения, выполняются перед резолюциями с более длинными предложениями. Эта стратегия скорее относится к упорядочиванию, а не к очищению, так как она меняет лишь порядок выполнения, но ни одну из возможных резолюций не исключает из рассмотрения.

*4.2.4.2 Факторизация*

Размер предложений по длине можно уменьшить, используя подстановки, так что некоторые литералы в предложении становятся одинаковыми. Эта операция называется факторизацией. Например:

###### C = {A(x, f(k)), A(b, y), A(a, f(x)), A(x, z)}

можно факторизовать подстановкой:

*θ =(b/x, f(k)/y, f(b)/z).*

Тогда получим:

*Cθ = {A(b, f(k)), A(a, f(b)), A(b, f(b))}.*

*Cθ* называется факториалом предложения *C*. Фактор предложения не обязательно единственный. Другой фактор:

*π = (a/x, f(k)/y, f(a)/z),*

*Cπ = {A(a, f(k)), A(b, f(k)), A(a, f(a)}.*

*4.2.4.3 Использование подслучаев*

Для любой пары предложений *C, D ∈ S* предложение *C* называется подслучаем предложения *D,* если существует такая подстановка π, что *Cπ ⊆ D.* Например:

*C = {A(x)},*

*D = {A(b), P(x)},*

то подстановка

*π = (b/x)*

приведет к

*Cπ = {A(b)}.*

то означает сокращение литералов.

*4.2.4.4 Гиперрезолюция*

Можно делать так, чтобы в резолюции участвовали сразу по несколько предложений. Это называется гиперрезолюцией. Предположим, что для конечного множества предложений *{C1, …, Cn}* и единственного предложения *B* удовлетворяются следующие условия:

1. *B* содержит *n* литералов *L1, …, Ln.*
2. Для каждого *i, 1≤ i ≤ n,* предложение *Ci,* содержит литерал , но не содержит дополнений никакого другого литерала из *B* и никакого литерала из любого предложения *Cj, j ≠ i.*

Множество *Sa = {Ci} ∪ {B}* будем называть конфликтом, а предложение:



его гиперрезольвентой. *Ra* можно вывести из *Sa.*

В большинстве случаев к конфликту приходят после соответствующих подстановок. Иными словами, для заданного множества предложений *Sa*, не удовлетворяющего определению конфликта, может найтись такая подстановка *π,* что *Saπ* будет конфликтом. Тогда *Sa* называется скрытым конфликтом.

В качестве примера гиперрезолюции рассмотрим множества:



Подстановка π*=(a/x, b/y)* дает



*Saπ -* конфликт с резольвентой , и *Sa* – скрытый конфликт.

*4.2.4.5С – упорядочение*

С – упорядочение предполагает линейность, так как при его определении различаются левое и правое родительские предложения. Пусть *С* – предложение из *S.* Обозначим через *[C]* предложение *C* с заданным на нем произвольным порядком литералов, а через *[S]* – множество таких упорядоченных предложений. Если предложение *[C]* порождается в линейном выводе  то пусть *[Ci-1]* и *[Bi-1]* будут его левым и правым предложениями с литералами предложения *Ci-1,* расположенными в порядке  (где  т.е. самый правый литерал левого родительского выражения является литералом резолюции), и с литералами предложения *Bi-1* в порядке . Ясно, что для  некоторого *i (i =1÷m).* Тогда упорядоченное предложение *Ci* таково:



т.е. литералы правого родительского предложения добавляются к литералам левого, при этом литералы резолюции, естественно, опускаются, а в случае повторения литералов сохраняются те, что расположены слева. Резолюция допускается только с самым правым литералом предложения *Ci.*

*Пример:*

**







*Y*



*Z,X*

Z





Y







Z

# *Компьютерный практикум*

Реализовать алгоритм C – упорядочивания.