Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ
УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ (ТУСУР)

УТВЕРЖДАЮ
Зав. кафедрой ЭМИС
_____И.Г. Боровской
«___»_____ 2017 г.

Н.Ю. Истомина, А.А. Матолыгин

ИНТЕЛЛЕКТУАЛЬНЫЕ СИСТЕМЫ УПРАВЛЕНИЯ ПРОЕКТАМИ

Методические указания для проведения практических занятий и самостоятельной работы для студентов

Томск, 2017

Н.Ю. Истомина, А.А. Матолыгин, ИНТЕЛЛЕКТУАЛЬНЫЕ СИСТЕМЫ УПРАВЛЕНИЯ ПРОЕКТАМИ// Методические указания для проведения практических занятий и самостоятельной работы для студентов, обучающихся по направлению 09.04.02 «Информационные системы и технологии» - Томск: Изд-во ТУСУР, 2017. – 101 с.

В пособии рассматриваются вопросы связанные с приобретением навыков и умений магистрантами по программированию на языке Visual Prolog и навыков работы с различными моделями знаний и построению экспертных систем.

Оглавление

Предварительные замечания

Настоящие рекомендации разработаны на основании требований Федерального Государственного образовательного стандарта высшего образования (ФГОС ВО) по направлению подготовки 09.04.02 «Информационные системы и технологии», утвержденного 30.10.2014 г. №1402, Положения о практиках студентов Томского государственного университета систем управления и радиоэлектроники, утвержденного приказом ректора ТУСУРа от 20.11.2014 г. Цель настоящих методических рекомендаций – помочь студентам в успешном прохождении обучения по курсу «Интеллектуальные системы управления проектами».

Изучение дисциплины призвано сформировать у магистрантов следующие общекультурные и профессиональные компетенции:

- способностью самостоятельно приобретать с помощью информационных технологий и использовать в практической деятельности новые знания и умения, в том числе в новых областях знаний, непосредственно не связанных со сферой деятельности (ОК-6);
- умением проводить разработку и исследование методик анализа, синтеза, оптимизации и прогнозирования качества процессов функционирования информационных систем и технологий (ПК-9);
- умением осуществлять моделирование процессов и объектов на базе стандартных пакетов автоматизированного проектирования и исследований (ПК-10).

По результатам выполнения каждой из практических работ должен быть оформлен отчет по практической работе согласно ОС ТУСУР 01-13. Разделы отчета располагаются в следующей последовательности.

- Титульный лист.
- Вид работы.
- Название работы.
- Цель работы.

- Основные теоретические разделы дисциплины необходимые для выполнения работы.
- Формулировка задания.
- Результаты выполнения индивидуального задания.
- Выводы по работе.

Практическая работа №1
«Выводы в логике высказываний»

Цель работы: закрепить знания по вопросам представления знаний на основе логической модели и вывода в ней.

Ознакомьтесь с основными конструкциями логики высказываний. Уясните правила построения формул в логике высказываний.

Обратите внимание на сходство и отличия в применении логических связок в естественном языке и в логических формулах. Приведите свои примеры сложных высказываний на естественном языке и переведите их в логические формулы.

Изучите бесскобочную форму записи логических формул.

Раскройте содержание понятия «интерпретация формулы логики высказываний», а также понятий «общезначимость», «противоречивость», «необщезначимость», «непротиворечивость», «выполнимость» формул логики высказываний. Изучите эти понятия на своих собственных примерах.

Ознакомьтесь с правилами эквивалентных преобразований формул. Обратите внимание на то, что эквивалентные преобразования действуют в обоих направлениях.

Разберитесь с понятием логического следствия. Выясните, как связаны между собой логическое следствие, общезначимость и противоречивость.

Изучите способы логического вывода в логике высказываний с точки зрения реализации их на компьютере.

Для определения вывода, при построение модели на основе логики высказываний, необходимо в тексте задания выделить простые предложения, обозначить их как атомы и затем представить каждое предложение в виде формулы. Например, имеем следующие утверждения:

- если солнце село в тучу (С), то завтра будет дождь (Д);
- солнце село в тучу.

Доказать, что следовательно, завтра будет дождь.

Формализуем эти утверждения. Две посылки и заключение будут представлены следующим образом:

С →Д

С

Д

Приведем все способы решения данной задачи.

Способ 1. Воспользуемся ОПРЕДЕЛЕНИЕМ логического следствия и таблицами истинности.

С	Д	С→Д	(С →Д)∧С
И	И	И	И
И	Л	Л	Л
Л	И	И	Л
Л	Л	И	Л

Конъюнкция посылок истинна только при одной интерпретации, заданной первой строкой, в этой же интерпретации истинным является и заключение (Д), следовательно, Д является логическим следствием посылок (С→Д) и (С).

Способ 2. Воспользуемся ТЕОРЕМОЙ 1 и таблицами истинности.

С	Д	С →Д	(С →Д)∧С	((С →Д)∧С) →Д
И	И	И	И	И
И	Л	Л	Л	И
Л	И	И	Л	И
Л	Л	И	Л	И

Так как формула ((С →Д)∧С) →Д общезначима, то Д является логическим следствием посылок (С →Д) и (С).

Способ 3. Воспользуемся ТЕОРЕМОЙ 2 и таблицами истинности.

С	Д	~Д	С Д	(С →Д) ∧С	(С →Д) ∧С∧~Д
И	И	Л	И	И	Л
И	Л	И	Л	Л	Л
Л	И	Л	И	Л	Л
Л	Л	И	И	Л	Л

Так как формула (С →Д) ∧С∧~Д противоречива, то Д является логическим следствием посылок (С →Д) и (С).

Способ 4. Воспользуемся ТЕОРЕМОЙ 1 и эквивалентными преобразованиями формул.

((С →Д)∧С) →Д = ((~С ∨Д)∧ С)→Д = ~((~С ∨Д) С) ∨Д = ((С ∧ ~Д) ∨ ~С) ∨Д =(С ∧ ~Д) ∨~С∨Д =(С ∨ ~С∨Д) ∧ (~С∨Д ∨ ~Д) =(■∨Д) ∧ (~С∨■) = (■∧■) =■.

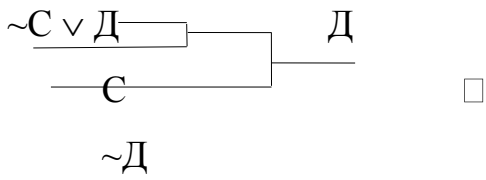
В данной формуле символ ■ означает общезначимую формулу. Так как в результате эквивалентных преобразований получена общезначимая формула, то Д является логическим следствием посылок.

Способ 5. Воспользуемся ТЕОРЕМОЙ 2 и эквивалентными преобразованиями формул.

((С →Д) ∧ С)∧~Д = ((~С ∨ Д) ∧С) ~Д = (~С ∨ Д) ∧С ∧ ~Д = (С ∧ ~С∧Д) ∨ (~С∧Д ∧ ~Д)= (□∧ ~Д) ∨ (С∧□)= □∨□= □.

В данной формуле символ □ означает противоречивую формулу. Так как в результате эквивалентных преобразований получена противоречивая формула, то Д является логическим следствием посылок.

Способ 6. Воспользуемся методом резолюций. Для доказательства приведем отрицание формулы к КНФ. Доказывать будем невыполнимость множества дизъюнктов, для этого необходимо, согласно ТЕОРЕМЕ 2, взять отрицание заключения.



Так как в процессе резолютивного вывода получен пустой дизъюнкт, то Д является логическим следствием посылок.

Задания.

Вариант 1

Если Степан не знал о необходимости декларировать доход, то он плохой законодатель. Если он знал и не декларировал, то он мошенник. Если Степан является плохим законодателем или мошенником, то ему нет места в Думе. Степан не декларировал свой доход. Следовательно, ему нет места в Думе. Доказать всеми возможными способами.

Вариант 2

Если исход скачек будет предрешен сговором или в игорных домах будут орудовать шулеры, то доходы от туризма упадут, и город пострадает. Если доходы от туризма упадут, полиция будет довольна. Полиция никогда не бывает довольна. Следовательно, исход скачек не предрешен сговором. Доказать всеми возможными способами.

Вариант 3

Если 6 – составное число, то 12 – составное число. Если 12 – составное число, то существует простое число, большее чем 12. Если существует простое число, большее чем 12, то существует составное число, большее, чем 12. Если 6 делится на 2, то 6 – составное число. 12 – составное число. Следовательно, 6 – составное число. Доказать всеми возможными способами.

Вариант 4

Контракт будет выполнен тогда и только тогда, когда дом будет закончен в феврале. Если дом будет закончен в феврале, то мы можем переезжать 1-го марта. Если мы не можем переезжать 1-го марта, то мы должны внести квартплату за март. Если контракт не будет выполнен, то мы должны внести квартплату за март. Следовательно, мы должны внести квартплату за март. Доказать всеми возможными способами.

Вариант 5

Если я пойду завтра на первое занятие, то должен буду встать рано, а если я пойду вечером на танцы, то лягу спать поздно. Если я лягу спать поздно, а встану рано, то я буду вынужден довольствоваться пятью часами сна. Я не могу довольствоваться пятью часами сна. Следовательно, я или не пойду завтра на первое занятие, или не пойду вечером на танцы. Доказать всеми возможными способами.

Вариант 6

Если Мери бросила Джона, то она уехала или в Россию, или в Израиль. Если Мери уехала в Россию, то ее арестовал КГБ. Если Мери уехала в Израиль, то ее арестовал Мосад. Мери не арестовал ни Мосад, ни КГБ. Значит Мери не бросила Джона. Доказать всеми возможными способами.

Вариант 7

Халиф Омар, сжегший Александрийскую библиотеку, рассуждал так: если ваши книги согласны с Кораном, то они излишни; если они не согласны с Кораном, то они вредны; но вредные или излишние книги следует уничтожать; значит, ваши книги следует уничтожить. Доказать правильность рассуждений халифа.

Вариант 8

Или Маша и Ваня одного возраста, или Маша старше Вани. Если Маша и Ваня одного возраста, то Наташа и Ваня не одного возраста. Если Маша старше Вани, то Ваня старше Пети. Следовательно, или Наташа и Ваня не одного возраста, или Ваня старше Пети. Доказать всеми возможными способами.

Вариант 9

Если я поеду автобусом, а автобус опоздает, то я пропущу назначенное свидание. Если я пропущу назначенное свидание и буду огорчен, то мне не следует ехать домой. Если я не получу эту работу, то я буду огорчен и мне следует поехать домой. Следовательно, если я поеду домой автобусом и автобус опоздает, то я получу эту работу. Доказать всеми возможными способами.

Вариант 10

Если завтра будет холодно, я надену шубу, если рукав будет починен. Завтра будет холодно, а рукав не будет починен. Следовательно, я не надену шубу. Доказать всеми возможными способами.

Практическая работа №2
«Введение в язык ПРОЛОГ. Простейшие программы»

Цель работы: изучение вопросов построения простейших программ на языке Пролог и изучение машины логического вывода на примере PIE Visual Prolog.

Простейшие программы.

Программирование в логических языках состоит в описании модели рассматриваемой предметной области. Модель строится в терминах объектов и отношений. Отношения между объектами могут быть заданы в виде фактов и правил. *Факт* – это некоторое утверждение, истинность которого считается доказанной.

Простейшая ПРОЛОГ-программа состоит из одного факта и имеет следующий вид:

***A*.**

что интерпретируется следующим образом: «утверждение ***A*** выполнимо», «факт ***A*** имеет место», «***A*** истинно». Точка в конце предложения обязательна и пролог программе выполняет роль дизъюнкции.

Обращение к ПРОЛОГ-программе, в классическом варианте и в машине вывода PIE Visual Prolog, организовано с помощью *вопросов*. Пример обращения:

? - *R*.

Интерпретация этого предложения такова: «выполнимо ли утверждение ***R*** ?», «имеет ли место факт ***R*** ?», «истинно ли ***R*** ?»

В ПРОЛОГе принята следующая концепция: «отсутствие факта означает его неудачное выполнение», поэтому, если программа состоит из одного факта «***A.***», а вопрос задан «***?- B.***», то ответом будет «нет» («неудача», «ложь», «No»), если же задан вопрос «***?- A***.», то ответом будет «да» («удача», «истина», «Yes»).

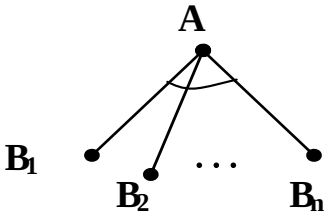
Синтаксически вопрос от факта отличается наличием вначале предложения конструкции «***?-*** ». Вопрос и факт обязательно заканчиваются точкой « . ». Утверждение без точки будем называть *целью*.

Следующий уровень сложности ПРОЛОГ-программы связан с рассмотрением новой конструкции языка, называемой *правилом***.** Правило имеет следующий вид:

$$\mathbf{A:- B_1, B_2, \dots , B_n.} \qquad (1)$$

Состоит правило из двух частей, разделенных конструкцией «**:-** », которую можно читать как «если». Левая часть правила («***A***») называется заголовком (головой предложения) и является атомарной формулой. Правая часть правила называется телом правила и представляет собой конечное множество целей, возможно пустое, здесь $\boldsymbol{B_1, B_2, \dots , B_n}$ – факты либо правила, соединенные в единую конструкцию символами « , ». Таким образом, символ « , » означает в теле правила связку «и» (конъюнкцию). Интерпретируется правило следующим образом: «утверждение ***A*** выполнимо, если выполнимы все $\boldsymbol{B_1, B_2, \dots , B_n}$», «заключение ***A*** имеет место, если имеют место одновременно и $\boldsymbol{B_1}$ ***и*** $\boldsymbol{B_2, \dots,}$ ***и*** $\boldsymbol{B_n}$», «***A*** истинно, если истинны все $\boldsymbol{B_1, B_2, \dots , B_n}$».

Графически правило (1) может быть представлено вершиной типа И:



Факт можно рассматривать как частный случай правила, когда тело правила пусто ($n = 0$).

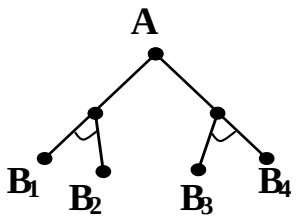
Используя понятие *процедуры*, ПРОЛОГ-программу, имеющую вид:

$$\mathbf{A:- B_1, B_2.}$$

$$\mathbf{A:- B_3, B_4.} \qquad (2)$$

можно прочитать следующим образом: если при выполнении процедуры ***A*** успешно выполнены процедуры $\boldsymbol{B_1}$ и $\boldsymbol{B_2}$, то процедура ***A*** выполнена успешно; если хотя бы одна из процедур $\boldsymbol{B_1}$ или $\boldsymbol{B_2}$ завершилась неуспешно,

то выполняется второе предложение процедуры ***A***, и соответственно выполняются процедуры B_3 и B_4; если обе эти процедуры закончены успешно, то процедура ***A*** успешно завершена, в противном случае выполнение процедуры ***A*** завершается неуспехом. Таким образом, правила в процедуре связаны связкой «или» (дизъюнкция). Графически правила (2) могут быть представлены в виде дерева с вершиной типа ИЛИ и вершинами типа И:



В Прологе принята стратегия поиска решения в глубину, когда ведется последовательный обход дерева И-ИЛИ сверху вниз и слева направо.

Задание.

Выделить в тексте простые предложения, обозначив их как атомы. Представить каждое утверждение в виде формулы на языке логики предикатов. На основе формул составить на языке Пролог программу, доказывающую утверждение текста. Представить на И-ИЛИ графе факты и поиск решения.

Пример 1.

Пусть имеются утверждения: «Овладеть языком программирования можно (PL), если работать с ним практически (W) и выучить его основы (F). Работа с языком практически возможна, если есть персональный компьютер (P). Выучить основы языка можно при условии работы в библиотеке (L). Выполнены условия работы в библиотеке и наличия персонального компьютера. Следовательно, можно овладеть языком программирования». ПРОЛОГ-программа, задающая описания приведенных выше утверждений, имеет следующий вид:

PL:- W, F.

W:- P.

F:- L.

L.

P.

Обращение к программе: **?- PL.,** даст ответ «да».

Допустим теперь, что условие работы в библиотеке выполнено, а персонального компьютера нет в наличии. ПРОЛОГ-программа в этом случае будет иметь следующий вид:

PL:- W, F.

W:- P.

F:- L.

L.

P:- fail.

Здесь автором пособия использован встроенный системный предикат ***fail***, который всегда приводит к безуспешному выполнению правила, в котором ***fail*** использован. Более привычная и наглядная запись отрицания в виде ***not(P)*** невозможна в силу языковых ограничений. Вопрос к программе: **?- PL.** даст ответ «нет».

Пример 2.

Халиф Омар, сжегший Александрийскую библиотеку, рассуждал так: «если ваши книги согласны с Кораном (A), то они излишни (E); если ваши книги не согласны с Кораном (A), то они вредны (H); но вредные или излишние книги следует уничтожать (D); значит, ваши книги следует уничтожить».

На языке логики предикатов данные рассуждения можно представить следующим образом:

$$\frac{A \to E, \quad \neg A \to H, \quad (E \vee H) \to D \quad A \vee \neg A}{D}$$

ПРОЛОГ-программа будет иметь следующий вид:

E:- A.

H:- not(A).

D:- H.

D:- E.

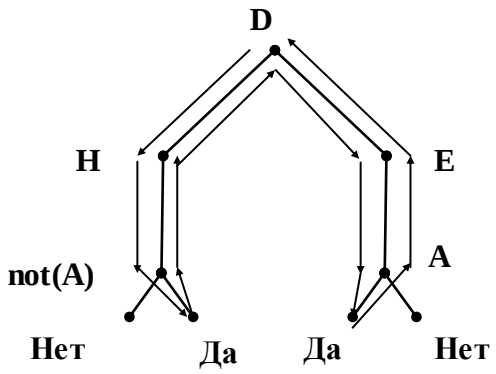
A.

A:- fail.

На вопрос о том, уничтожать ли Александрийскую библиотеку:

?- D., следует ответ **Да.**

На графе И-ИЛИ программа и поиск решения представляются следующим образом:



Вариант 1:

Если 9 марта будет тепло, то Джон поедет в Сан-Франциско или Лас-Вегас. Кейт поедет туда же, куда и Джон. Если Мери поедет в Лас-Вегас, то и Джон поедет в Лас-Вегас. Если Мери не поедет в Лас-Вегас, то Джон поедет в Сан-Франциско. Если 8 марта будет холодно, то 9 марта будет тепло. Если 8 марта будет холодно, то Мери не поедет в Лас-Вегас. 8 марта будет холодно.

Вопрос: поедет ли Кейт в Сан-Франциско?

Вариант 2:

Никакой сладкоежка не откажется от вкусного торта. Некоторые люди, которые отказываются от вкусного торта, не любят сладкого.

Справедливо ли утверждение: некоторые люди, не любящие сладкого, не являются сладкоежками.

Вариант 3:

Шар 2 находится всегда в том месте, где находится шар 1. Шар 3 находится в месте А. Если шар 3 находится в месте А, то шар 1 находится в месте В. Где находится шар 2?

Вариант 4:

Если спрос больше предложения, то цена на данный товар возрастает. Когда цена растет и на данный товар есть заменители, покупатели берут товары-заменители. Когда покупатели берут товары-заменители, спрос на данный товар падает. Спрос больше предложения. Для данного товара есть товары-заменители.

Вопрос: упадет ли спрос на товар?

Вариант 5:

Если команда А выигрывает в футбол, то город А1 торжествует, а если выигрывает команда В, то торжествовать будет город В1. Выигрывает или А или В. Однако если выигрывает А, то город В1 не торжествует. А если выигрывает в, то не будет торжествовать город А1. Следовательно, город В1 будет торжествовать тогда и только тогда, когда не будет торжествовать город А1.

Вариант 6:

Любой студент хочет закончить институт. Некоторые студенты обладают особыми способностями.

Доказать: студенты, обладающие особыми способностями, хотят закончить институт.

Вариант 7:

Сегодня тучи. Если сегодня тучи, то будет дождь. Если будет дождь, то вырастут грибы.

Доказать: вырастут грибы.

Вариант 8:

Если не работает лифт, я пойду по лестнице пешком. Лифт не работает. Если я пойду пешком по лестнице, то я не куплю стол.

Доказать: куплю ли я стол?

Вариант 9:

Если некто бизнесмен, то он любит считать деньги. Если он любит считать деньги, то деньги у него есть. Олег мужчина. Если он мужчина, то у него черная машина. Если у него есть деньги, то у него дорогая машина. Олег бизнесмен. Если дорогая машина, то Феррари. Если он выберет черную машину, то это будет или Феррари, или Волга.

Доказать: у Олега черная Феррари.

Вариант 10:

Все первокурсники встречаются со всеми второкурсниками. Ни один первокурсник не встречается ни с одним студентом предпоследнего курса. Существуют первокурсники. Следовательно, ни один второкурсник не является студентом предпоследнего курса.

На самостоятельное изучение предлагаются следующие задания.

1. Изучить вопрос построения проекта в Visual Prolog (http://wikiru.visual-prolog.com/Index.php?title=Visual_Prolog_%D0%B4%D0%BB%D1%8F_%D1%87%D0%B0%D0%B9%D0%BD%D0%B8%D0%BA%D0%BE%D0%B2)
2. Из примеров системы программирования Visual Prolog собрать и скомпилировать машину вывода PIE Visual Prolog. Решить представленную задачу с помощью машины вывода.

Практическая работа №3
«Выводы в логике предикатов»

Цель работы: закрепить знания по вопросам представления знаний на основе логической модели и вывода в ней.

Разберитесь с синтаксисом логики предикатов. Уясните основные синтаксические отличия логики высказываний и логики предикатов.

Раскройте содержание понятия «интерпретация формулы логики предикатов первого порядка», обратите внимание на отличия в правилах интерпретации формул логики предикатов и формул логики высказываний.

Изучите способ получения универсальной области интерпретации формул логики предикатов. Разберитесь с тем, как может быть использована теорема Эрбрана в логическом выводе.

Уясните принципиальные отличия в использовании теоремы Эрбрана и метода резолюций в логическом выводе.

Процедура Эрбрана предполагает генерацию основных примеров дизъюнктов из заданного множества. Для генерации основных примеров необходимо множество констант, а таким множеством является эрбрановский универсум. Если на множестве основных примеров существует хотя бы одно невыполнимое, то заданное множество дизъюнктов невыполнимо. Например, пусть S = {P(x), ~P(f(a))}, тогда эрбрановский универсум H□ = {a, f(a), f(f(a)), f(f(f(a))), ... }, если вместо переменной x подставить f(a), то получим невыполнимое множество основных примеров S'= {P(f(a)), ~P(f(a))}, а это, в свою очередь, означает, что множество S невыполнимо.

Для вывода в логике предикатов необходимо выделить простые предложения, обозначить их как атомы и затем представить каждое предложение в виде формулы. Пример, рассмотрим следующие утверждения:

F1: Каждый, кто хранит деньги в банке, получает проценты.

G: Если не выплачиваются проценты, то никто не хранит деньги в банке.

Выделим и обозначим в тексте простые предложения:

S(x, y) – x хранит y в банке;

M(x) – x - деньги;

I(x) – x - проценты;

P(x, y) – x получает y.

F1: $(\forall x)((\exists y)(S(x, y) \wedge M(y)) \rightarrow (\exists z)(I(z) \wedge P(x, z)))$.

G: $\sim(\exists z)I(z) \rightarrow (\forall x)(\forall y)(S(x, y) \rightarrow \sim M(y))$.

Переведем посылку в дизъюнкты:

$(\forall x)((\exists y)(S(x, y) \wedge M(y)) \rightarrow (\exists z)(I(z) \wedge P(x, z))) = (\forall x)(\sim(\exists y)(S(x, y) \wedge \wedge M(y)) \vee (\exists z)(I(z) \wedge P(x, z))) = (\forall x)((\forall y)(\sim S(x, y) \vee \sim M(y)) \vee (\exists z)(I(z) \wedge P(x, z))) = (\forall x)(\forall y)(\exists z)(\sim S(x, y) \vee \sim M(y) \vee (I(z) \wedge P(x, z)))$;

$(\forall x)(\forall y)(\exists z)(\sim S(x, y) \vee \sim M(y) \vee (I(z) \wedge P(x, z)))$ – предваренная нормальная форма;

$(\forall x)(\forall y)(\exists z)((\sim S(x, y) \vee \sim M(y) \vee I(z)) \wedge(\sim S(x, y) \vee \sim M(y) \vee P(x, z))) = =(\forall x)(\forall y)((\sim S(x, y) \vee \sim M(y) \vee I(f(x, y))) \wedge(\sim S(x, y) \vee \sim M(y) \vee P(x, f(x, y))))$ – сколемовская стандартная форма;

дизъюнкты, полученные из стандартной формы:

$\sim S(x, y) \vee \sim M(y) \vee I(f(x, y))$, $\sim S(x, y) \vee \sim M(y) \vee P(x, f(x, y))$.

Переведем отрицание заключения в дизъюнкты,

$\sim(\sim(\exists z)I(z) \rightarrow (\forall x)(\forall y)(S(x, y) \rightarrow \sim M(y))) = \sim((\exists z)I(z) \vee (\forall x)(\forall y)(\sim S(x, y) \vee \sim M(y)))$

$= \sim((\exists z)(\forall x)(\forall y)(I(z) \vee \sim S(x, y) \vee \sim M(y))) = (\forall z)(\exists x)(\exists y)(\sim I(z) \wedge S(x, y) \wedge M(y))$ – предваренная нормальная форма;

$(\forall z)(\sim I(z) \wedge S(g(z), h(z)) \wedge M(h(z)))$ – сколемовская стандартная форма;

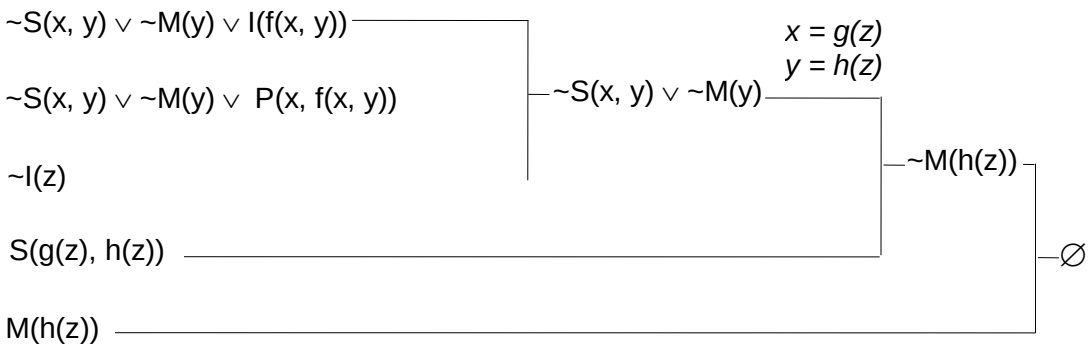
дизъюнкты, полученные из стандартной формы:

$\sim I(z)$,

$S(g(z), h(z))$,

$M(h(z))$.

Резолютивный вывод на дизъюнктах, получены из посылки и отрицания заключения, имеет следующий вид:



Получен пустой дизъюнкт, заключение доказано.

Задания.

Вариант 1

Используя процедуру Эрбрана, доказать невыполнимость множества дизъюнктов S = {P(x, a, g(x, b)), ~P(f(y), z, g(f(a), b))}.

Ни один человек не является четвероногим. Все женщины – люди. Следовательно, ни одна женщина не является четвероногой. Доказать.

Вариант 2

Используя процедуру Эрбрана, доказать невыполнимость множества дизъюнктов S = {P(x), Q(x, f(x)) ∨ ~P(x), ~Q(g(y), z)}.

Каждый член комитета богат и демократ. Некоторые члены комитета – старики. Следовательно, существуют старики-де-мократы. Доказать.

Вариант 3

Используя процедуру Эрбрана, доказать невыполнимость множества дизъюнктов S = {P(x), ~P(x) ∨ Q(x, a), ~Q(y,a)}.

Некоторые республиканцы любят всех демократов. Ни один республиканец не любит ни одного социалиста. Следовательно, ни один демократ не является социалистом. Доказать.

Вариант 4

Используя процедуру Эрбрана, доказать невыполнимость множества дизъюнктов S = {~P(x) ∨ Q(x, f(x)), P(x), ~Q(g(z), y)}.

Ни один первокурсник не любит второкурсников. Все, живущие на шестом этаже, – второкурсники. Следовательно, ни один первокурсник не любит никого из живущих на шестом этаже. Доказать.

Вариант 5

Используя процедуру Эрбрана, доказать невыполнимость множества дизъюнктов

$$S = \{\sim C(x) \vee W(x), \sim C(x) \vee R(x), C(a) , O(a), \sim O(x) \vee \sim R(x)\}.$$

Ни один торговец наркотиками не является наркоманом. Некоторые наркоманы привлекались к ответственности. Следовательно, некоторые люди, привлекавшиеся к ответственности, не являются торговцами наркотиками. Доказать.

Вариант 6

Используя процедуру Эрбрана, доказать невыполнимость множества дизъюнктов

$$S = \{P(x) \vee Q(x, f(x)), \sim P(x), \sim Q(g(y), z)\}.$$

Студенты суть граждане. Следовательно, голоса студентов суть голоса граждан. Доказать.

Вариант 7

Используя процедуру Эрбрана, доказать невыполнимость множества дизъюнктов

$$S = \{P(a), \sim D(y) \vee L(a, y), \sim P(x) \vee \sim Q(y) \vee \sim L(x, y), D(b) , Q(b)\}.$$

Никакой торговец подержанными автомобилями не покупает подержанный автомобиль для своей семьи. Некоторые люди, покупающие подержанные автомобили для своих семей, – жулики. Следовательно, некоторые жулики не являются торговцами подержанными автомобилями. Доказать.

Вариант 8

Используя процедуру Эрбрана, доказать невыполнимость множества дизъюнктов

$$S = \{\sim S(y) \vee \sim C(y), S(b), V(a, b), \sim C(z) \vee V(a, z)\}.$$

Некоторые пациенты любят своих докторов. Ни один пациент не любит знахаря. Следовательно, никакой доктор не является знахарем. Доказать.

Вариант 9

Используя процедуру Эрбрана, доказать невыполнимость множества дизъюнктов

$S = \{\sim S(x, y) \vee \sim M(y) \vee I(f(x)), \sim S(x, y) \vee \sim M(y) \vee E(x, f(x)), \sim I(z), S(a, b), M(b)\}$.

Все первокурсники встречаются со всеми второкурсниками. Ни один первокурсник не встречается ни с одним студентом предпоследнего курса. Существуют первокурсники. Следовательно, ни один второкурсник не является студентом предпоследнего курса. Доказать.

Вариант 10

Используя процедуру Эрбрана, доказать невыполнимость множества дизъюнктов

$S = \{\sim E(x) \vee V(x) \vee S(x, f(x)), \sim E(x) \vee V(x) \vee \vee C(f(x)), P(a), E(a), \sim S(a, y) \vee P(y),$

$\sim P(x) \vee \sim V(x), \sim P(x) \vee \sim C(x)\}$.

Боб – мальчик, у которого нет автомобиля. Джейн любит только тех мальчиков, у которых есть автомобили. Следовательно, Джейн не любит Боба. Доказать.

Практическая работа №4
«Типы предикатов. Типовые задачи»

Цель работы: знакомство с синтаксисом и получение практических навыков составления простейших программ на языке логического программирования – ПРОЛОГ.

Переменные и константы.

Исходными элементами описания предметной области в приведенных выше примерах являются отдельные утверждения: простые – факты и сложные – правила. При этом не рассматривались состав и структура утверждения в смысле связи действия, агента, объекта и других грамматических составляющих предложения. Однако не всегда возможно адекватно описать предметную область таким простым способом. Для углубленного анализа и описания предметной области язык программирования должен иметь возможность работы с константами, переменными и сложными структурами. ПРОЛОГ обладает такими возможностями.

Рассмотрим следующий факт:

«Том родитель Боба».

Сказуемым или предикатом в данном предложении является «родитель», этот предикат связывает два объекта «Том» и «Боб». В языках логического программирования для записи подобных фактов существует стандартная форма записи:

предикат(объект_1, объект_2).

Тогда наш факт может быть записан следующим образом:

parent(tom,bob).

Объекты «Том» и «Боб» являются здесь константами, отношение с именем «parent» – это предикат. Резольвента – это множество целей, которые необходимо выполнить, чтобы получить ответ на поставленный вопрос.

Большинство ПРОЛОГ-трансляторов работают только с латинским алфавитом в качестве имен предикатов, поэтому для записи имен предикатов используется латинская нотация.

Совокупность фактов называется базой данных. Этот термин будет использоваться для объединения фактов при описании предметной области для решения конкретной задачи. Продолжив перечисление родственных отношений, получим некоторую базу следующего вида:

parent(pam,bob).

parent(tom,bob).

parent(tom,liz).

parent(bob,ann).

parent(bob,pat).

parent(pat,john).

Пусть имеется вопрос: «является ли Лиз родителем Боба? »

В терминах выбранной нотации ПРОЛОГа этот вопрос выглядит так:

?- parent(liz, bob).

Ответ «нет». Получила ПРОЛОГ-система его следующим образом:

из вопроса формируется резольвента, в нашем случае – parent(liz, bob);

выбирается первая цель из резольвенты (она у нас единственная);

просматривается база данных в попытке согласовать выбранную цель и факт из базы, в нашем случае такая попытка оказывается безуспешной.

Таким образом, для того чтобы получить положительный ответ на поставленный вопрос, база данных должна содержать факт, удовлетворяющий следующим условиям:

имя факта и вопроса должны совпадать;

должно совпадать число аргументов в вопросе и факте;

вопрос и факт должны иметь одни и те же константы на тех же местах.

При выполнении этих условий вопрос удовлетворяет факту и, следовательно, ответ положителен. В нашем случае не нашлось факта, удовлетворяющего вопросу.

Теперь, положим, необходимо знать, родителем кого является Пэт. Перебор всех вопросов с указанием всех имен родственников, до получения ответа «да» – это не лучший вариант решения этой проблемы. Эту задачу можно решить, сформулировав вопрос следующим образом:

?- parent(pat, X).

Литера x означает в вопросе неизвестный заранее объект. Такого рода объекты называются переменными. Переменная в ПРОЛОГе – это последовательность букв и цифр, начинающаяся с прописной буквы или символа подчеркивания и содержащая только символы букв, цифр и подчеркивания. Переменная, состоящая из одного символа подчеркивания, – это анонимная переменная, использующаяся в предложении только один раз. В процессе поиска ответа на вопрос ПРОЛОГ-система просматривает базу данных и пытается найти эквивалентную замену переменной x объектом из базы, используя при этом алгоритм унификации. Ответом на вопрос: ?- parent(pat, X) будет X =john.

Переформулируем предыдущую задачу следующим образом: определить, кто родитель Джона. На вопрос

?- parent(X, john).

система ответит X=pat. Таким образом, отношение parent ПРОЛОГ-система может использовать различными способами: если известны оба его аргумента, то проверяется выполнимость данного отношения на этих аргументах; если неизвестен один из аргументов, неважно какой из них, то система пытается найти эквивалентную замену переменной объектом из базы данных.

Во многих версиях ПРОЛОГа приняты следующие соглашения: заглавные буквы обозначают переменные, а строчные буквы, целые или действительные числа, или любые символы, заключенные в кавычки, обозначают постоянные значения – константы.

Предположим, что необходимо ответить на вопрос: «кто является родителем родителя Джона?» («кто дед Джона?»). В ПРОЛОГе это можно

сделать в одном вопросе, воспользовавшись общей переменной и соединив оба вопроса, используя операцию конъюнкции: ?- parent (Y, john), parent(x, y). Этот вопрос читается следующим образом: «Существуют ли такие x и y, что parent(y, john) и parent(x, y) выполнимы одновременно?»

Рассмотрим действия ПРОЛОГ-системы при выполнении данного вопроса:

формируется резольвента из двух целей: parent (y, john), parent(x, y);

выбирается первая цель из резольвенты – parent (y, john);

просматривается база данных в попытке согласовать выбранную цель и факт из базы. Попытка оказывается успешной, в базе найден факт parent(pat,john);

переменная y с этого момента конкретизирована и ей соответствует значение pat;

первая цель резольвенты выполнена, система переходит к следующей цели – parent (x, pat);

просматривается база данных в попытке согласовать выбранную цель и факт из базы, попытка оказывается успешной, в базе найден факт parent(bob,pat);

переменной x присваивается значение bob;

в резольвенте отсутствуют невыполненные цели, значит, вопрос успешно выполнен, система выдает ответ: x = bob, y = pat.

Фундаментальными операциями над переменными является конкретизация и унификация, с помощью которых выполняются однократное присваивание и передача параметров.

Сформулируем предыдущую задачу в общем виде:

«Определить кто чей предок».

Запрос к базе данных оформим в виде следующего правила:

predok(x, y):- parent (x, z), parent (z, y).

Представленное правило на естественном языке читается следующим образом: «если x – родитель z, и z – родитель y, то x – предок (дед) y ».

Правило определяет предикат predok, имеющий два аргумента. Предикат predok(x, y) истинен, если установлено взаимно-однозначное соответствие между именем ребенка z (parent (x, z)) и именем родителя z (parent (z, y)).

Предыдущий вопрос будет записан следующим образом:

?- predok(x, john).

Действия ПРОЛОГ-системы здесь следующие:

формируется резольвента predok(x, john);

выбирается цель из резольвенты, т.е. predok(x, john);

делается попытка согласовать выбранную цель и заголовок правила из базы, попытка оказывается успешной;

переменная y с этого момента конкретизирована, и ей соответствует значение john, переменная x не конкретизирована;

формируется новая резольвента parent(x, z), parent(z, john);

дальнейшие действия совпадают с действиями ПРОЛОГ-системы при решении предыдущей задачи с той лишь разницей, что вместо переменной X здесь используется Z.

Основы системы Visual Prolog.

Отличия между Visual Prolog и традиционным Прологом.

Различия между традиционным Прологом и Visual Prolog можно провести по следующим категориям:

Различия в структуре программы: Различия между Visual Prolog и традиционным Прологом имеются, но они не существенны. Они сводятся к пониманию того, как различаются декларации (declarations) от определений (definitions), а также к явному выделению цели Goal. Все это поддержано специальными ключевыми словами.

Файловая структура программы: Visual Prolog предоставляет возможности структуризации программ с использованием файлов различного типа.

Границы видимости: программа системы Visual Prolog может поддерживать функционирование, располагающееся в различных модулях с использованием концепции идентификация пространств.

Объектная ориентированность: программа на языке Visual Prolog может быть написана как объектно-ориентированная программа с использованием классических свойств объектно-ориентированной парадигмы.

Различия в структуре программ.

Декларации и Определения.

В Прологе, когда необходимо использовать предикат, то это делается без каких-либо предварительных указаний движку Пролога о таких намерениях. Например, в предыдущих руководствах клауз предиката grandFather (дедушка) был непосредственно записан с использованием традиционной для Пролога конструкции голова-тело.

Аналогично, когда в традиционном Прологе требуется использовать составной домен, можно его использовать без предупреждения движка по поводу этого намерения. Просто используется домен тогда, когда в этом возникает необходимость.

Однако, в Visual Prolog, перед написанием кода для тела клауза предиката необходимо сначала объявить о существовании такого предиката компилятору. Аналогично, перед использованием любых доменов они должны быть объявлены и представлены компилятору.

Причиной такой необходимости в предупреждениях является попытка как можно раньше обнаружить возможность исключений периода исполнения.

Под "исключениями периода исполнения (*runtime exceptions*), понимаются события, возникающие только во время исполнения программы. Например, если использовать целое число в качестве аргумента функтора, а вместо этого по ошибке использовали вещественное число, то в процессе исполнения возникла бы ошибка периода исполнения (в программах,

написанных для ряда компиляторов, но не для Visual Prolog) и программа в этом случае завершилась бы неуспешно.

Когда объявляются предикаты или доменты, которые определены, то появляется своего рода позиционная грамматика (какому домену принадлежит какой аргумент), доступная компилятору. Более того, когда Visual Prolog выполняет компиляцию, он тщательно проверяет программу на наличие таких грамматических ошибок, наряду с другими.

Благодяря этому свойству Visual Prolog, повышается конечная эффективность программиста. Программист не должен ждать, когда реально работающая программа совершит ошибку. Часто конкретная последовательность событий, приводящая к исключению периода исполнения кажется настолько неуловимой, что ошибка может проявиться через много лет, или она может заявить о себе в критической ситуации и привести к непредсказуемым последствиям!

Все это автоматически ведет к тому, что компилятор должен получать точные инструкции по поводу предикатов и доменов в виде соответствующих объявлений, которые должны предшествовать определениям.

Ключевые слова.

Программа на Visual Prolog, представляемая кодом, разделяется ключевыми словами на секции разного вида путем использования ключевых слов, предписывающих компилятору, какой код генерировать. К примеру, есть ключевые слова, обозначающие различие между декларациями и определениями предикатов и доменов. Обычно, каждой секции предшествует ключевое слово. Ключевых слов, обозначающих окончание секции, нет. Наличие другого ключевого слова обозначает окончание предыдущей секции и начало другой.

Исключением из этого правила являются ключевые слова **implement** и **end implement**. Код, содержащийся между этими ключевыми словами, есть код, который относится к конкретному классу. Те, кто не понимает

концепцию класса, может пока (в рамках этого руководства) представлять себе его как модуль или раздел какой-то программы более высокого уровня.

В рамках этого руководства представлена только часть ключевых слов, приведенных ниже. В Visual Prolog есть и другие ключевые слова, они упоминаются в других рукодствах.

В этом руководстве используются следующие ключевые слова:

implement и **end implement**

Среди всех ключевых слов, обсуждаемых здесь, это единственные ключевые слова, используемые парно. Visual Prolog рассматривает код, помещенный между этими ключевыми словами, как код, принадлежащий одному классу. За ключевым словом **implement** обязательно ДОЛЖНО следовать имя класса.

open

Это ключевое слово используется для расширения области видимости класса. Оно должно быть помещено вначале кода класса, сразу после ключевого слова **implement** (с именем класса и, возможно именем интерфейса - прим. перев.).

constants

Это ключевое слово используется для обозначения секции кода, которая определяет неоднократно используемые значения, применяемые в коде. Например, если строковый литерал "PDC Prolog" предполагается использовать в различных местах кода, тогда можно единожды определить мнемоническое (краткое, легко запоминаемое слово) для использования в таких местах:

```
constants
pdc="PDC Prolog".
```

Заметьте, что определение константы завершается точкой (.). В отличие от переменных Пролога константы должны начинаться со строчной буквы (нижний регистр).

domains

Это ключевое слово используется для обозначения секции, объявляющей домены, которые будут использованы в коде. Синтаксис таких объявлений позволяет порождать множество вариантов объявлений доменов, используемых в тексте программы.

Вообще говоря, объявляется функтор, который будет использоваться в качестве домена и ряд доменов, которые используются в качестве его аргументов. Функторы и составные домены рассматриваются в соответствующем руководстве.

class facts

Это ключевое слово представляет секцию, в которой объявляются факты, которые будут в дальнейшем использоваться в тексте программы. Каждый факт объявляется как имя, используемое для обозначения факта, и набор аргументов, каждый из которых должен соответствовать либо стандартному (предопределенному), либо объявленному домену.

class predicates

Эта секция содержит объявления предикатов, которые определяются в тексте программы в разделе clauses. И опять, объявление предиката - это имя, которое присваивается предикату, и набор аргументов, каждый из которых должен соответствовать либо стандартному (предопределенному), либо объявленному домену.

clauses

Среди всех разделов, существующих в тексте программ на Visual Prolog, это единственный раздел, который близко совпадает с традиционными программами на Прологе. Он содержит конкретные определения объявленных в разделе **class predicates** предикатов, причем синтаксически им соответствующие.

goal

Этот раздел определяет главную точку входа в программу на языке системы Visual Prolog. Более детальное описание приводится ниже.

Goal (цель)

В традиционном Прологе, как только какой-либо предикат определен в тексте, Пролог-машине может быть предписано выполнение программы, начиная с этого предиката. В Visual Prolog это не так. Будучи компилятором, он отвечает за генерацию эффективного исполняемого кода написанной программы. Этот код исполняется не в то же время, когда компилятор порождает код. Поэтому компилятору надо знать заранее точно, с какого предиката начнется исполнение программы. Благодаря этому, когда программа вызывается на исполнение, она начинает работу с правильной начальной точки. Как можно уже догадаться, откомпилированная программа может работать независимо от компилятора Visual Prolog и без участия среды IDE.

Для того, чтобы это стало возможным, введен специальный раздел, обозначенный ключевым словом *Goal*. Его можно представлять как особый предикат, не имеющий аргументов. Это предикат - именно тот предикат, с которого вся программа начинает исполняться.

Файловое структурирование программ

Необходимость помещения всех частей большой программы в один файл несомненно является неудобством. Это ведет к тому, что программа становится нечитаемой и иногда неправильной. Visual Prolog предусматривает возможность деления текста программы на отдельные файлы, используя среду IDE (Integrated Development Environment) и, кроме того, IDE позволяет помещать логически связанные фрагменты текста программы в отдельные файлы. Несмотря на такое разделение программы на части, сущности, находящиеся в общем использовании, доступны.

Например, если имеется домен, который используется несколькими файлами, то объявление домена делается в отдельном файле и к этому файлу из других файлов существует доступ.

Для упрощения этого руководства мы будем использовать один файл для разработки текста программы. На самом деле в процессе построения

программы среда IDE создавала бы автоматически несколько файлов, но мы сейчас будем это умалчивать. Об этом написано в других руководствах.

Расширение Области Видимости

В Visual Prolog текст программы разделен на отдельные части, какждая часть определяется как класс (class). В объектно-ориентированных языках программирования, класс - это пакет кода и ассоциированные с ним данные. Это требует длительных объяснений и сделано в других руководствах. Те, кто не знаком с объектно-ориентированным программированием, может представлять себе класс нестрого как синоним понятия модуль. В Visual Prolog каждый класс обычно помещается в отдельный файл.

В процессе исполнения программы, часто бывает так, что программе может потребоваться вызвать предикат, который определен в другом классе (файле). Аналогично, данные (константы) или домены могут быть востребованы в другом файле.

Visual Prolog позволяет делать такие взаимные ссылки на предикаты или данные используя так называемую концепцию области видимости (*scope access*). Это может стать понятным на примере. Предположим имеется предикат, называемый **pred1** и определенный в классе называемом **class1** (который помещается в другом файле, согласно стратегии среды IDE), и мы хотим вызвать этот предикат в теле клауза некоторого другого предиката **pred2**, находящегося в другом файле (скажем, **class2**). Тогда вот как предикат **pred1** должен бы быть вызван в теле клауза предиката **pred2** :

```
clauses
  pred3:-
    ...
    !.

  pred2:-
    class1::pred1, % pred1 неизвестен в этом файле.
                   % Он определен в другом файле,
                   % поэтому требуется квалификатор класса.
    pred3,
    ...
```

В приведенном примере видно, что тело клауза предиката **pred2** вызывает два предиката **pred1** и **pred3**. Поскольку **pred1** определен в другом файле (где определен класс **class1**), постольку слово **class1** с символом :: (два двоеточия) предшествует слову **pred1**. Это можно назвать как квалификатор класса. Но предикат **pred3** определен внутри того же самого файла, что и предикат **pred2**. Поэтому здесь нет необходимости использовать **class2::** перед вызовом предиката **pred3**.

Технически такое поведение объясняется следующим: область видимости (**scope visibility**) предиката **pred3** находится внутри той же области, где находится предикат **pred2**. Поэтому здесь не нужно показывать, что **pred3** и **pred2** находятся в одном классе. Компилятор автоматически увидит определение предиката **pred3** внутри той же области, в классе **class2**.

Область видимости конкретного класса лежит внутри границ, где класс определен (то есть в интервале между ключевыми словами **implement - end implement**). Предикаты, определенные здесь, могут вызывать друг друга без квалификатора класса и двойного двоеточия (::).

Область видимости класса может быть расширена использованием ключевого слова open. Это ключевое слово информирует компилятор о том, что в этот класс должны быть "доставлены" имена (предикатов / констант / доменов), которые были определены в других файлах. Если область видимости расширена, то необходимости использования квалификатора класса (с двойным двоеточием) нет.

```
open class1
  ...

clauses
  pred3:-
    ...
    !.

  pred2:-
    pred1, % Внимание: квалификатор "class1::" больше не нужен,
         % поскольку область видимости расширена
         % использованием ключевого слова 'open'
```

```
  pred3,
  ...
```

Объектная ориентированность

Visual Prolog является полностью объектно-ориентированным языком.

Весь код, который разрабатывается для программы, помещается в класс. Это происходит само собой, даже если Вы не интересуетесь объектными свойствами языка. Это свойство обнаруживается в примере, разбираемом в этом руководстве. Код помещается в класс, называемый "family1", несмотря на то, что мы не используем объекты, порождаемые этим классом. Кроме того, вы этом классе используются общедоступные предикаты, находящиеся в других классах.

Рассмотрим следующий пример. Текст программы приведен ниже. Его надо поместить в файл с именем family1.pro. Реальный текст программы создадим с помощью среды IDE (Integrated Development Environment) системы Visual Prolog. На самом деле файлов больше, чем требуется для этого руководства, но они будут созданы (и в дальнейшем будут поддерживаться) автоматически средой IDE. Пошаговая инструкция по использованию IDE (с фрагментами экрана) для разработки программы приводится ниже.

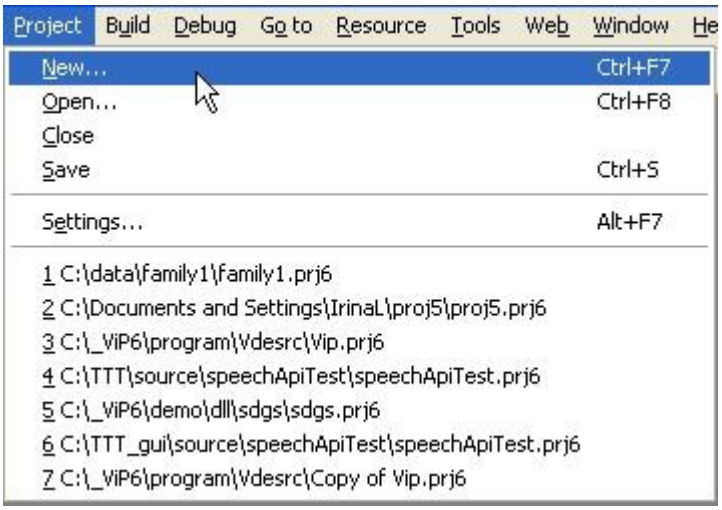
Но прежде всего ознакомимся с главным текстом программы:

```
implement family1
open core

constants
  className = "family1".
  classVersion = "$JustDate: $$Revision: $".

clauses
  classInfo(className, classVersion).

domains
  gender = female(); male().

class facts - familyDB
  person : (string Name, gender Gender).
  parent : (string Person, string Parent).

class predicates
  father : (string Person, string Father) nondeterm anyflow.
```

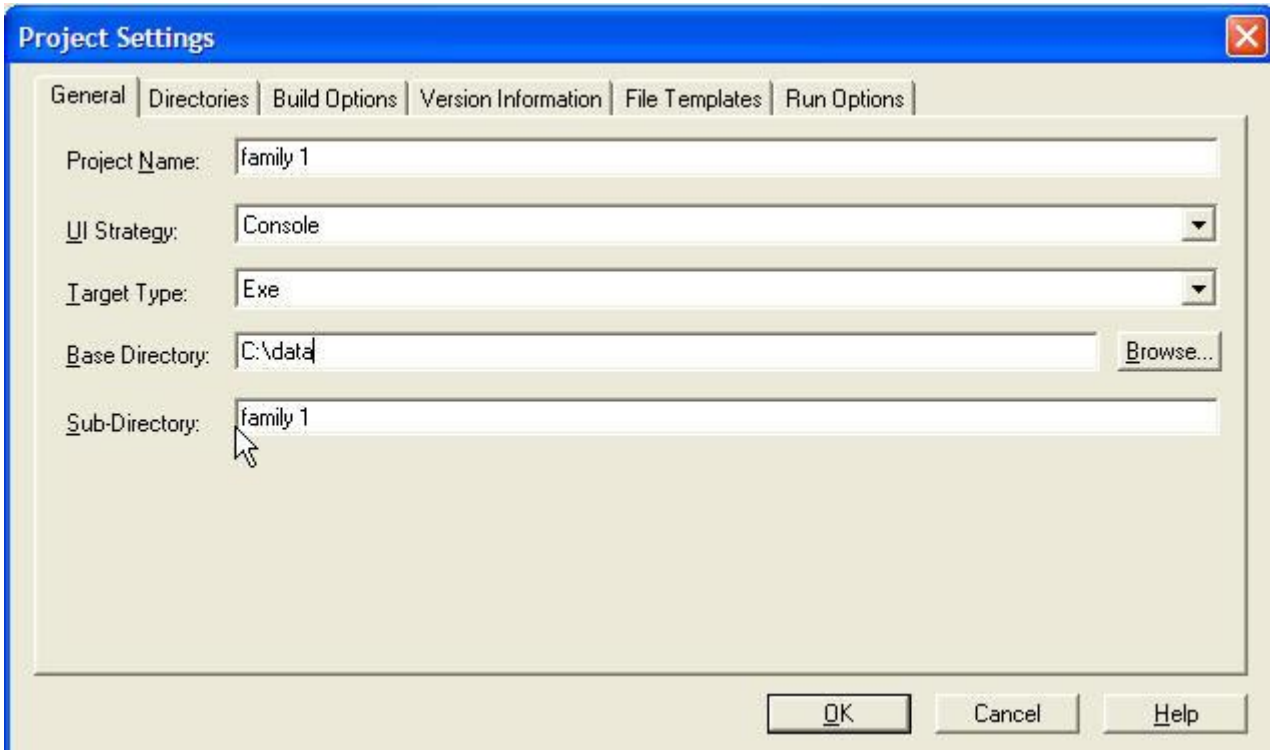
```
clauses
  father(Person, Father) :-
    parent(Person, Father),
    person(Father, male()).

class predicates
  grandFather : (string Person, string GrandFather) nondeterm anyflow.
clauses
  grandFather(Person, GrandFather) :-
    parent(Person, Parent),
    father(Parent, GrandFather).

class predicates
  ancestor : (string Person, string Ancestor) nondeterm anyflow.
clauses
  ancestor(Person, Ancestor) :-
    parent(Person, Ancestor).
  ancestor(Person, Ancestor) :-
    parent(Person, P1),
    ancestor(P1, Ancestor).

class predicates
  reconsult : (string FileName).
clauses
  reconsult(FileName) :-
    retractFactDB( familyDB),
    file::consult(FileName, familyDB).

clauses
  run():-
    console::init(),
    stdIO::write("Load data\n"),
    reconsult("fa.txt"),
    stdIO::write("\nfather test\n"),
    father(X, Y),
      stdIO::writef("% is the father of %\n", Y, X),
    fail.
  run():-
    stdIO::write("\ngrandFather test\n"),
    grandFather(X, Y),
      stdIO::writef("% is the grandfather of %\n", Y, X),
    fail.
  run():-
    stdIO::write("\nancestor of Pam test\n"),
    X = "Pam",
    ancestor(X, Y),
      stdIO::writef("% is the ancestor of %\n", Y, X),
    fail.
  run():-
    stdIO::write("End of test\n").
end implement family1

goal
mainExe::run(family1::run).
```

Шаг 1: Создайте в IDE новый консольный проект

Шаг 1a. После старта среды программирования Visual Prolog, выберите New из меню Project.



Шаг 1b. Появляется диалог. Введите соответствующую информацию. Удостоверьтесь, что стратегия пользовательского интерфейса (UI Strategy) Console, а НЕ GUI.



Шаг 2: Постройте пустой проект

Шаг 2a. Когда проект только что создан, среда будет показывать проектное окно, как показано ниже. В этот момент пока никакой информации

о том, от каких файлов зависит проект, нет. Однако основные тексты программ проектных файлов уже созданы.

Шаг 2b. Выберите из меню Build позицию Build. Пустой проект, который ничего не делает, будет построен. (В этот момент времени никаких сообщений о ошибках быть не должно).

В процессе построения проекта посмотрите на сообщения, которые динамически появляются в окне Messages среды IDE. (Если окно Messages не видно, его можно вызвать на передний план, используя меню Window в среде IDE). Вы заметите, что среда IDE разумно включит в Ваш проект ВСЕ в необходимые модули PFC (Prolog Foundation Classes). Классы PFC являются теми классами, которые содержат поддержку функционирования программ, которые Вы строите.

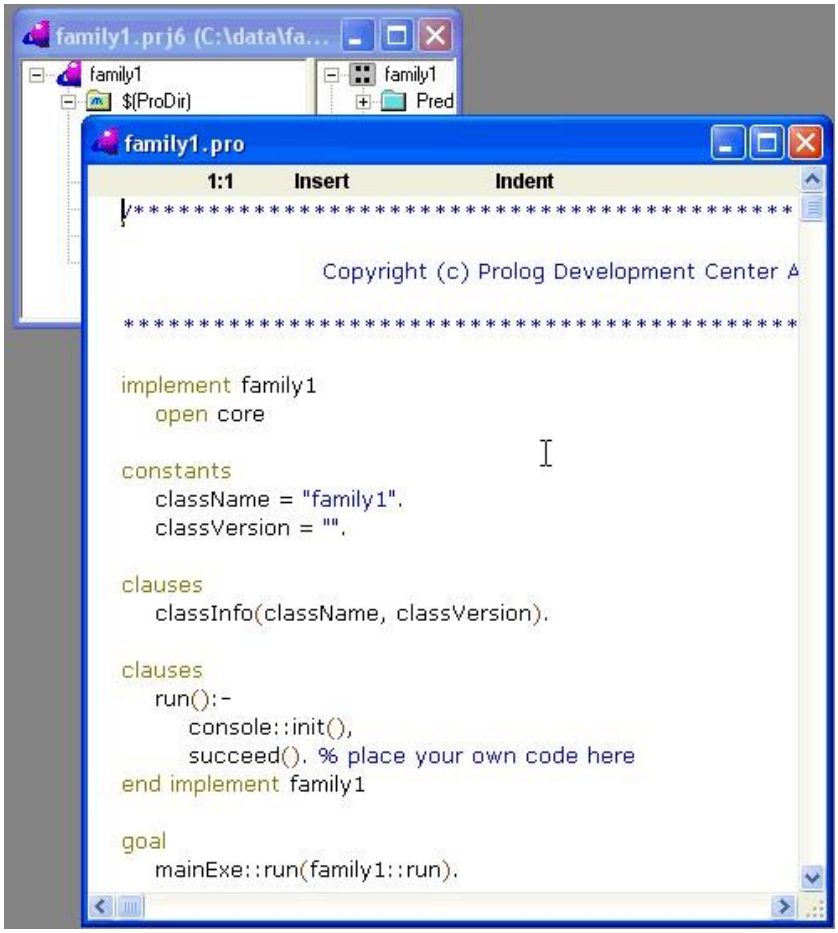
Шаг 3: Поместите в файл family1.pro актуальный код

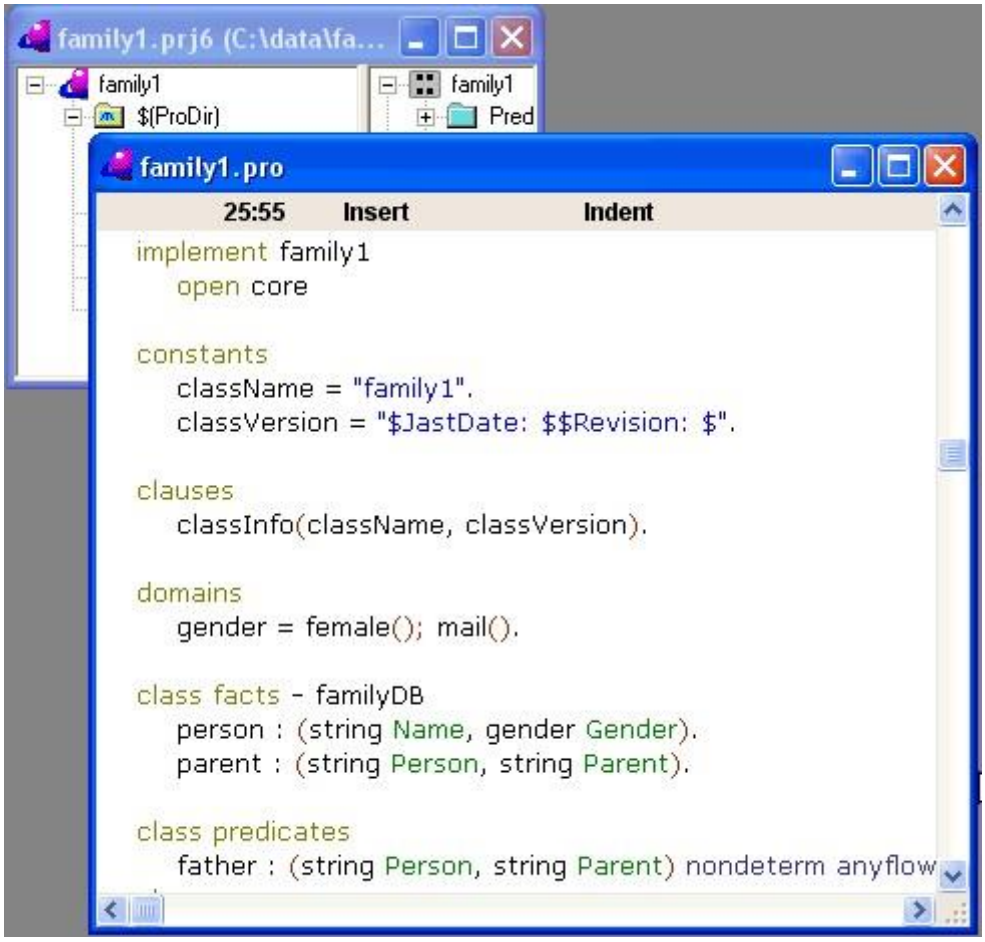
Шаг 3a. Начальный код, созданный самой средой IDE имеет самый общий вид и сам по себе не делает ничего полезного. Вам необходимо удалить этот код полностью и скопировать (через Copy и Paste) полностью (актуальный на данный момент) код программы family1.pro (приведенный в этом руководстве) в окно редактора.

Шаг 3b. После выполнения операции copy-paste окошко с текстом файла family1.pro должно выглядеть так, как показано ниже.

Шаг 4: Перестроение кода проекта

Повторите вызов команды меню Build. IDE теперь уведомит, что исходный текст поменялся, и выполнит все необходимые действия для перекомпиляции соответствующих разделов. Если Вы вызовите команду меню Rebuild All, тогда ВСЕ модули проекта будут перекомпилированы. В случае больших проектов это может занимать значительное время. Rebuild All обычно используется в качестве финальной фазы после ряда небольших


family1.prj6 (C:\data\fa...
family1
$(ProDir)
family1
Pred
family1.pro
1:1
Insert
Indent
Copyright (c) Prolog Development Center A
implement family1
open core
constants
className = "family1".
classVersion = "".
clauses
classInfo(className, classVersion).
clauses
run():-
console::init(),
succeed(). % place your own code here
end implement family1
goal
mainExe::run(family1::run).



изменений и соответствующих компиляций, дабы удостовериться в том, что все в полном порядке.

В процессе выполнения команды Build среда IDE выполняет не только компиляцию. В это же время выясняется, нужны ли проекту другие модули из набора PFC (Prolog Faundation Classes), и, если это так, то такие модули добавляются и вызывается повторная перекомпиляция, если необходимо. Этот процесс можно наблюдать по сообщениям, появляющимся в окне сообщений (Message Window). Можно увидеть как IDE вызвало построение проекта дважды, поскольку были обнаружены директивы "include", повлиявшие на это.

Файл:FundamentalVisualProlog7.jpg

Шаг 5: Исполнение Программы

Теперь у нас есть приложение, успешно откомпилированное с помощью Visual Prolog. Для того, чтобы проверить работу приложения, которое только что построено, можно запустить ее выполнение, вызвав

команду из меню Build | Run in Window. Но в нашем случае это приведет к ошибке. Причина заключается в том, что наша маленькая программа для обеспечения своей работы пытается читать текстовый файл fa.txt. Этот текстовый файл содержит содержит записи относительно лиц, которые (записи) программа должна обработать. Вопросов синтаксита представления данных в этом файле мы коснемся позже.

Сейчас нам необходимо написать этот текстовый файл с использованием текстового редактора и поместить его в ту же директорию, где располагается сама исполняемая программа. Обычно исполняемая программа помещается в поддиректории проектной директории, имеющей имя EXE.

Давайте создадим такой файл сами с использованием среды IDE. Вызовите команду меню File | New. Появляется окно создания новой сущности Create Project Item. Выберите Текстовый файл (Text File) в списке слева. Затем выберите директорию, где будет размещаться файл (та же директория, где располагается исполняемое приложение family1.exe). Присвойте теперь имя fa.txt файлу и нажмите кнопку Create (создать). До тех пор, пока имя файла не введено, кнопка Create (создать) будет неактивна (надпись серого цвета). Удостоверьтесь, что флажот Add to project as module (добавить в проект на правах модуля) включен. Полезность включения файла в проект заключается в том, что это будет удобно для последующей отладки и других действий.

Файл следует заполнить текстом следующего содержания:

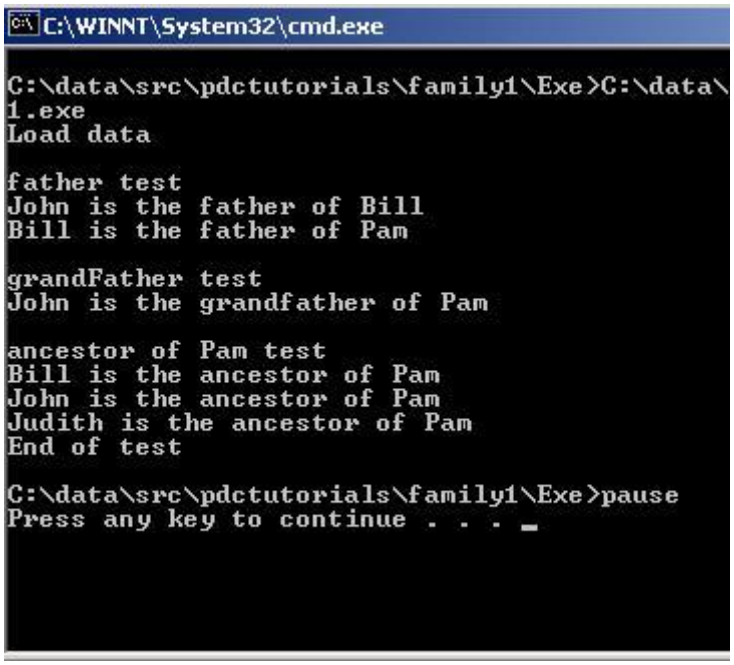
```
clauses
   person("Judith",female()).
   person("Bill",male()).
   person("John",male()).
   person("Pam",female()).
   parent("John","Judith").
   parent("Bill","John").
   parent("Pam","Bill").
```

Несмотря на то, что это файл данных, Вы заметите, что синтаксис файла очень похож на обычный код Пролога! Даже несмотря на то, что

программа теперь откомпилирована и представлена в двоичном формате, Вы можете менять ход и результаты ее исполнения, изменяя данные, которые программа обрабатывает. И эти данные записываются в этот текстовый файл с использованием того же самого синтаксиса Пролога. Visual Prolog таким образом эмулирует свойство использования динамически изменяемого кода, характерное для традиционного пролога. Хотя не все свойства использования динамически изменяемого кода традиционного Пролога воспроизводятся, но достаточно сложные структуры данных (или не сложные, как в этом примере) могут быть использованы.

Синтаксис файла fa.txt ДОЛЖЕН быть совместим с определениями доменов в программе. Например, функтор, применяемый для определения персоны ДОЛЖЕН быть person и никак иначе. В противном случае соответствующий составной домен не будет инициализирован. (Понятия функторов и составных доменов рассматриваются в других руководствах).

Теперь, запустив программу по команде меню Build | Run in the Window вы увидите:



Программа обрабатывает данные, помещенные в файл fa.txt и порождает логические связи персон, данные о которых там помещены.

Рассмотрение программы

Логика программы очень проста. Мы ее уже обсуждали, где рассматривалось как корректное представление данных может помочь построить эффективную программу на Прологе. Давайте теперь обратим внимание на логику используемого метода.

При старте непосредственно будет вызван главный предикат. Здесь все построена вокруг предиката **run**. Предикат **run** прежде всего зачитывает данные, записанные в файле fa.txt. После загрузки данных, программа систематически переходит от одной проверки к другой, обрабатывая данные, и выводит на консоль выводы на базе этих проверок.

Обработка данных основывается на стандартных механизмах отката и рекурсивных вызовов. В этом руководстве детально не рассматривается как работает Пролог, но приведем краткое описание работы.

Когда предикат **run** вызывает предикат father, он не ограничивается первым удовлетворяющим решением предиката father. Применение предиката **fail** в конце последовательности понуждает механизм Пролога к поиску следующего прохода предиката father. Такое поведение, называется **backtracking** (откат), поскольку кажется, что механизм Пролога буквально перепрыгивает назад, минуя ранее исполненный код.

Это происходит рекурсивно (то есть как повторяющийся или циклический процесс) и в каждом цикле предикат father вырабатывает новый результат. В итоге все возможные определения "отцов", представленные данными исчерпываются и у предиката **run** нет другого выхода как перейти к следующему своему клаузу.

Предикат father (так же как и ряд других предикатов) объявлен как нетерминированный. Для обозначения недетерминированных предикатов используется ключевое слово **nondeterm**, обозначающее, что предикат может вырабатывать множество решений посредством **бэктрекинг**а (отката).

Если никакое ключевое слово в объявлении предиката не используется, то предикат получает режим процедуры, которая может выработать только одно решение, и предикат father прекратил бы работу после получения

первого же результата и не смог бы быть вызван повторно. Заметим, следует быть очень осторожными в использовании отсечения (cut), обозначаемого как !, в клаузах таких недетерминированных предикатов. Если в конце клаузы предиката father помещено отсечение, то предикат выработает только одно решение (даже если он объявлен как недетерминированный).

Квалификатор направлений ввода-вывода **anyflow** говорит компилятору о том, что параметры, назначенные предикату могут быть как связанными, так и свободными, без каких либо ограничений. Это означает, что используя одну и ту же декларацию предиката father, можно будет получать как имя отца (в случае, если имя потомка связано) так и имя потомка (в случае, если имя отца связано). Ситуация, когда оба параметра связаны, также обрабатывается - и в этом случае проверяется существует ли отношение между данными, представленными параметрами.

И, наконец, квалификатор **anyflow** включает ситуацию, когда оба параметра предиката father являются свободными переменными. В этом случае, предикат father вернет в ответ на запрос различные комбинации отношений родитель-потомок, предусмотренные в программе (представленные в загружаемом файле fa.txt). Последний вариант как раз и использован в предикате **run**, как видно во фрагменте, приведенном ниже. Можно заметить, что переменные X и Y переданные предикату father не связаны при его вызове.

```
clauses
    run():-
        console::init(),
        stdIO::write("Load  data\n"),
        reconsult("fa.txt"),
        stdIO::write("\nfather test\n"),
        father(X,Y),
        stdIO::writef("% is the father of %\n", Y, X),
        fail.
```

Задание.

На основе представленного примера программы определить ближайших родственников: братьев, сестер, тетей, дядей, племянников,

внуков, двоюродных братьев и сестер, внучатых племянников. Исходную базу фактов расширить для получения удачных исходов для всех предикатов.

Практическая работа №5
«Выводы в продукционной модели»

Цель работы: закрепить знания по вопросам представления знаний на основе продукционной модели и вывода в ней.

Рассмотрите варианты задания продукционного правила. Обратите внимание на сходство и отличие в определении правил.

Изучите структуру продукционной системы и способы получения вывода в системе. Выясните, какие функции выполняет каждый компонент системы.

Рассмотрите два способа вывода в продукционных системах: прямой и обратный. Придумайте свой пример продукционной системы, разберите прямой и обратный вывод в данной системе.

Ознакомьтесь со способами визуального представления правил в продукционных системах. Представьте правила Вашей продукционной системы в виде И/ИЛИ-графа.

Раскройте содержание понятия «конфликтный набор». Изучите способы разрешения конфликтов в продукционных системах в зависимости от типа вывода. Приведите примеры.

Рассмотрите особенности архитектуры продукционной системы, использующей для вывода модель доски объявлений.

Назовите сильные и слабые стороны продукционных систем.

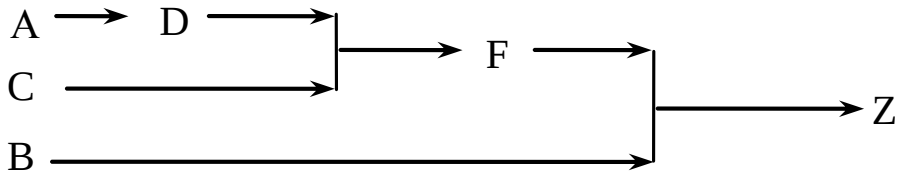
Пусть база правил в продукционной системе имеет содержимое:

если F и B то Z; если C и D то F; если A то D;

рабочая память: A, B, H, C.

Рассмотрим, каким образом «работают» правила. Система построена так, что один раз выбранное правило из базы правил выполняться будет только один раз. Оно как бы «выгорает». Первым выгорает правило «если A то D», так как A уже имеется в базе данных. В качестве следствия этого правила получается логический вывод о наличии ситуации D, которая

заносится в рабочую область. Это вызывает выгорание правила «если C и D то F», и, как следствие, выводится ситуация F, и она заносится в базу данных. Это, в свою очередь, вызывает выгорание правила «если F и B то Z» с занесением Z в базу данных. Такой способ называется прямым выводом. Графически вывод можно изобразить следующим образом:



Задания.

База правил и рабочая память в продукционной системе имеет содержимое, заданное в вариантах. Проиллюстрировать графически механизм прямого и обратного логического вывода факта A. Обратите внимание на изменение содержимого рабочей памяти в процессе вывода. Проведите упорядочение правил вывода. Рассмотрите возможные конфликты при прямом и обратном выводе.

Вариант 1. База правил:

если F и D и E то B; если G то C; если B и C то A; если R то D; если S то A;

если F и G то M;

рабочая память: G, E, R, F.

Вариант 2. База правил:

если B и C и D то A; если E то B; если G и H то C; если F то B; если E то A;

рабочая память: G, H, D, F.

Вариант 3. База правил:

если B и C и D, то A; если F и G, то B; если H и D, то E; если E, то A;

рабочая память: G, H, D, F.

Вариант 4. База правил:

если C и D то B; если E то B; если F и G то E; если B то A; если H то C;

рабочая память: G, H, D, F.

Вариант 5. База правил:

если D то B; если F и H то D; если B и C то A; если G и R то E; если E то B; если F и G то C;

рабочая память: G, H, F.

Вариант 6. База правил:

если B то A; если E и F и D то B; если G то C; если H то C; если C то A; если D то H;

рабочая память: G, H, D.

Вариант 7. База правил:

если B то A; если E и F и G то C; если H то D; если C то B; если D то B; если E то D;

рабочая память: G, H, F.

Вариант 8. База правил:

если B и C то A; если E и D то A; если H то C; если R то D; если G то E; если F и G то B;

рабочая память: G, H, R.

Вариант 9. База правил:

если B то A; если F то B; если G то F; если C то A; если H то E; если D и E то B;

рабочая память: G, H, D.

Вариант 10. База правил:

если B и C, то A; если D, то B; если E, то B; если F то, C; если G то C;

рабочая память: E, F, G.

Практическая работа №6
«Циклы и повторения»

Цель работы: освоить основные методы построения циклических участков кода.

Предопределенных конструкций повторения в Прологе **нет.** Для организации повторяющихся участков кода используется рекурсия. Как известно из теории алгоритмов рекурсивная функция определяется следующим образом. Рассмотрим случай функции одного переменного. Пусть имеется функция *f(x)* и некоторая константа *a* и функция двух переменных *g(x, y).* Тогда если выполняется

$$f(const) = a$$
$$f(x+1) = g(x, f(x))$$

функция *f* является рекурсивной.

Первое правило можно использовать как правило остановки цикла, второе как средство вычисления последующих значений. Для остановки поиска последующих решений следует использовать отсечение. В этом случае, вы ставите восклицательный знак в хвосте предложения Хорна. После того, как система найдет восклицательный знак, она прерывает поиск новых решений. Восклицательный знак называется cut – отсечение. Давайте проиллюстрируем использование отсечений очень популярным алгоритмом среди программистов на Прологе. Например написать предикат, который находит факториал числа. Математики определяют факториал как:

factorial(0)→1
factorial(n)→n×factorial(n-1)

Используя хорновские предложения, это определение становится следующим:

```
fact(N, 1) :- N<1, !.
fact(N, N*F1) :- fact(N-1, F1).
```

Отсечение предотвращает попытку Пролога применить второе предложение для N=0.

```
implement facfun

open core, console

class predicates

fact : (integer, integer) procedure (i,o).

clauses

classInfo("facfun", "1.0").

fact(N, 1) :- N<1, !.

fact(N, N*F1) :- fact(N-1, F1).

run() :- console::init(), fact(read(), F), write(F), nl.

end implement facfun

        goal mainExe::run(facfun::run).
```

Задания.

1. Дано натуральное число n. Вычислить:
$$1\cdot 2+2\cdot 3\cdot 4+3\cdot 4\cdot 5\cdot 6+\ldots+n(n+1)\ldots 2n.$$
2. Вычислить:
$$\cfrac{1}{1+\cfrac{1}{3+\cfrac{1}{5+\cfrac{1}{7+\cfrac{1}{\ddots\,+\cfrac{1}{101+\cfrac{1}{103}}}}}}}$$
3. Дано действительное число $x\neq 0$. Вычислить:
$$\cfrac{x}{x^2+\cfrac{2}{x^2+\cfrac{4}{x^2+\cfrac{8}{x^2+\cfrac{16}{\ddots\,+\cfrac{256}{x^2}}}}}}$$
4. Для заданных значений n и x вычислить выражение:
$$s=\sin x+\sin\sin x+\ldots+\underbrace{\sin\sin\ldots\sin x}_{n\ раз}$$
5. Найти сумму $2^2+2^3+\ldots+2^{10}$ без возведения в степень.
6. Дано натуральное число n. Найти сумму $n^2+(n+1)^2+\ldots+(2n)^2$.

7. Найти сумму $-1^2+2^2-3^2+4^2-\ldots+10^2$. Операторами принятия решения пользоваться запрещается.
8. Найти сумму кубов всех целых чисел от *a* до *b*. Значения *a* и *b* вводятся с клавиатуры
9. Найти среднее арифметическое квадратов всех целых чисел от *a* до *b*. Значения *a* и *b* вводятся с клавиатуры (*b*>*a*).
10. Вычислить сумму $1+1/2+1/3+\ldots+1/n$.
11. Вычислить сумму $2/3+3/4+4/5+\ldots+10/11$.
12. Вычислить:

$$\sum_{i=1}^{100}\frac{1}{i^2}$$

13. Вычислить:

$$\sum_{i=1}^{50}\frac{1}{i^3}$$

14. Вычислить:

$$\sum_{i=1}^{10}\frac{1}{i!}$$

15. Вычислить:

$$\sum_{i=1}^{128}\frac{1}{(2i)^2}$$

16. Вычислить:

$$\prod_{i=1}^{52}\frac{i+1}{i+2}$$

17. Дано натуральное n, действительное число x. Вычислить:

$$\sum_{i=1}^{n}\frac{x^i}{i!}$$

18. Дано натуральное n, действительное число x. Вычислить:

$$\sum_{i=1}^{n}\frac{x+\cos(ix)}{2^i}$$

19. Дано натуральное n, действительное число x. Вычислить:

$$\sum_{i=1}^{n}\left(\frac{1}{i!}+\sqrt{|x|}\right)$$

Практическая работа №7
«Выводы в семантических сетях»

Цель работы: закрепить знания по вопросам представления знаний на основе семантической сети и вывода в ней.

Раскройте содержание понятия «семантика». Дайте определение синтаксическим, семантическим и прагматическим отношениям, принятым в семиотике.

Разберитесь с тем, что принято называть семантической сетью. Обратите внимание на целостность образа сети и неразделимость синтаксических и семантико-прагматических знаний о внешнем мире.

Изучите TLC-модель. Назовите основные отношения, принятые в данной модели. Определите достоинства и недостатки TLC-модели.

Разберитесь с применением первичного и вторичного определения понятия при представлении разных видов информации в виде понятийных структур.

Рассмотрите средства, дающие возможность представлять в сети события и действия. Обратите внимание на два основных уровня языка, принятых в лингвистике: уровень поверхностных структур и уровень глубинных структур. Приведите пример.

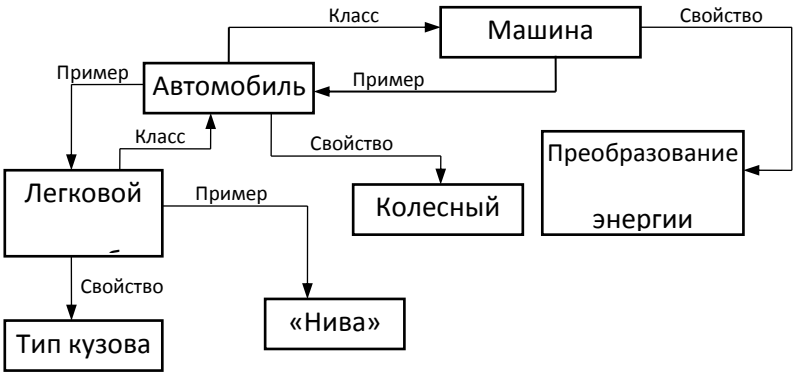
Изучите способы вывода в семантических сетях. Выясните, на каких отношениях определен механизм наследования.

Уясните механизм вывода в семантических сетях, основанный на построении подсети, соответствующей вопросу, и сопоставлении ее с базой знаний. Разберитесь со способом вывода, называемым перекрестным поиском.

Рассмотрите семантическую сеть специального вида, носящую название «функциональная семантическая сеть». Обратите внимание на то, как задаются параметры, участвующие в решении задачи, и как задаются функциональные отношения, связывающие между собой эти параметры.

Разберитесь с механизмами вывода в функциональной семантической сети, основанными на распространяющихся волнах и паросочетаниях. Выявите достоинства и недостатки рассмотренных методов.

При построении TLC-модели необходимо определить взаимосвязи между понятиями. Построим TLC-модель понятия «автомобиль».



Задание. Постройте TLC-модель для определения понятия, заданного в варианте; поскольку слова, используемые в определении понятия, сами обозначают понятия, то, определив их, постройте некоторую структуру, определяющую каждое понятие через взаимосвязи с другими имеющимися понятиями. Рассмотрите не менее десяти понятий в сети.

Вариант 1. Понятие «студент».

Вариант 2. Понятие «профессор».

Вариант 3. Понятие «шкаф».

Вариант 4. Понятие «компьютер».

Вариант 5. Понятие «стол».

Вариант 6. Понятие «журнал».

Вариант 7. Понятие «книга».

Вариант 8. Понятие «ребенок».

Вариант 9. Понятие «трактор».

Вариант 10. Понятие «посуда».

Практическая работа №8
«Сложные термы. Списки»

Цель работы: освоить основные приемы работы со списками в языке Пролог.

Списки.

Обработка списков как обработка последовательности элементов - это мощная техника, используемая в Прологе. В этом руководстве рассматриваются вопросы что такое списки, как их объявлять, и затем приводятся несколько примеров, показывающих как использовать работу со списками в приложениях. Определены два хорошо известных предиката Пролога – member (член, элемент) и append (добавить) – с рассматрением обработки списков как с рекурсивной, так и процедурной точки зрения.

Затем представлен предикат findall - стандартный предикат языка системы Visual Prolog, позволяющий собирать решения в единую цель.

В Прологе список (list) является объектом, содержащим внутри произвольное число других объектов. Списки соответствуют, грубо говоря, массивам в других языках, но, в отличие от массивов, список не трубует декларирования его размера до начала его использования.

Список, содержащий числа 1, 2 и 3 записывается как

```
[ 1, 2, 3 ]
```

Порядок элементов в этом списке значим:

- Число "1" является первым элементом,
- "2" - второй,
- "3" - третий.

Список [1, 2, 3] и список [1, 3, 2] различны.

Каждый компонент списка называется элемент (*element*). Для того, чтобы сформировать списковую структуру данных, следует разделять элементы запятыми и заключать их всех в квадратные скобки. Посмотрим на некоторые примеры:

```
["dog", "cat", "canary"]
["valerie ann", "jennifer caitlin", "benjamin thomas"]
```

Один и тот же элемент может быть представлен в списке несколько раз, например:

```
[ 1, 2, 1, 3, 1 ]
```

Объявление Списков

Для объявления домена - списка целых используется декларация домена, как показано ниже:

```
domains
  integer_list = integer*.
```

Звездочка означает "список этого"; то есть, integer* означает "список целых". Обратите внимание на то, что слово "*list*" не имеет специального значения в Visual Prolog. Вы равным образом могли бы назвать Ваш списковый домен как zanzibar. Именно звездочка, а не имя, предписывает этому домену быть списком.

Элементами в списке может быть что угодно, включая другие списки. Но все элементы в списке должны принадлежать одному домену, и дополнительно к декларации спискового домена должна быть декларация **domains** для элементов:

```
domains
  element_list = elements*.
  elements = ....
```

Здесь *elements* должны быть приравнены к простым доменным типам (например, *integer, real* или *symbol*) или к набору возможных альтернатив, обозначенных различными функторами. Visual Prolog не допускает смешивание стандартных типов в списке. Например, следующие декларации ошибочно представляют списки, созданные из *integers, reals* и *symbols*:

```
element_list = elements*.
elements =
    integer;
    real;
    symbol.
       /* Неправильно */
```

Выходом для объявления списков из *integer*, *real* и *symbols* является объявление домена общего для всех типов, где функтор показывает какому типу принадлежит тот или иной элемент. Например:

```
element_list = elements*.
elements =
   i(integer);
   r(real);
   s(symbol).
      /* функторами являются i, r и s */
```

Головы и Хвосты

Список на самом деле является рекурсивным составным объетом. Он состоит из двух частей - головы списка, которым является первый элемент, и хвоста - списка, который включает все следующие элементы.

Хвост списка всегда есть список; голова списка есть элемент.

Например,

```
голова списка [a, b, c] есть a
хвост списка [a, b, c] есть [b, c]
```

Что происходит, когда мы имеем дело со списком, содержащим один элемент? Ответом является:

```
головой списка [c] является c
хвостом списка [c] является []
```

Если многократно отнимать первый элемент от хвоста списка, мы получим в конечном итоге пустой список ([ ]).

Пустой список не может быть разбит на голову и хвост.

Это означает, что, концептуально говоря, списки имеют древовидную структуру подобно другим составным объектам. Древовидная структура списка [a, b, c, d] есть:

```
      list
     /    \
    a    list
        /    \
       b    list
           /    \
          c    list
              /    \
             d      []
```

Более того, одноэлементный список, такой как [a] - это не тот же самый элемент, который этот список содержит, поскольку [a] является действительно составной структурой данных, как это видно здесь:

```
    list
   /    \
  a     []
```

Представления Списков.

Пролог содержит метод для явного обозначения головы и хвоста списка. Вместо разделения элементов запятыми можно отделять голову от хвоста вертикальной чертой (|). Например,

[a, b, c] эквивалентно [a|[b, c]]

и, продолжая процесс,

[a|[b,c]] эквивалентно [a|[b|[c]]],

что эквивалентно [a|[b|[c|[]]]]

Можно даже использовать оба способа разделения в одном и том же списке, рассматривая вертикальную черту как разделитель самого низкого уровня. Следовательно, можно записать [a, b, c, d] как [a, b|[c, d]]. Таблица 1 дает дополнительные примеры.

Таблица 1: Головы и Хвосты списков

Список	Голова	Хвост
['a', 'b', 'c']	'a'	['b', 'c']
['a']	'a'	[]
/*пустой список*/ [ ]	неопределен	неопределен
[[1, 2, 3], [2, 3, 4], []]	[1, 2, 3]	[[2, 3, 4], []]

В Таблице 2 приведены некоторые примеры унификации списков.

Таблица 2: Унификация Списков

Список 1	Список 2	Связывание Переменных
[X, Y, Z]	[эгберт, ест, мороженое]	X=эгберт, Y=ест, Z=мороженое

[7]	[X | Y]	X=7, Y=[]
[1, 2, 3, 4]	[X, Y | Z]	X=1, Y=2, Z=[3,4]
[1, 2]	[3 | X]	fail

Использование Списков

Поскольку списки являются в действительности рекурсивными составными структурами данных, для их обработки необходимы и рекурсивные алгоритмы. Самый естественный способ обработки списков - сквозной просмотр, в ходе которого что-то делается с каждым элементом, до тех пор, пока не достигнут конец.

Как правило, такого рода алгоритмы используют два клауза. Один из них говорит о том, как поступать с обыкновенным списком, который может быть разделен на голову и хвост. Другой говорит о том, что делать с пустым списком.

Вывод Списков на печать

Например, если Вы хотите только вывести на печать элементы списка, то вот что Вы делаете:

```
class my
predicates
  write_a_list : (integer*).
end class

implement my
clauses
  write_a_list([]). /* Если список пустой, ничего не делаем. */
  write_a_list([H|T]):- /* Сопоставляем голову с H и хвост с T, и... */
    stdio::write(H),stdio::nl, /*выводим H и переводим строку*/
    write_a_list(T).
end implement

goal
  console::init(),
  my::write_a_list([1, 2, 3]).
```

Здесь мы видим два клауза write_a_list, которые можно выразить но обычном языке.

1. Для вывода на печать пустого списка ничего не надо делать.

2. Иначе, для вывода на печать списка, вывести на печать его голову (она есть просто элемент), и потом вывести на печать хвост списка (он, как известно, есть список).

   Первый раз, когда вызывается:

```
my::write_a_list([1, 2, 3]).
```

такой вызов сопоставляется со вторым клаузом, с головой H=1 и T=[2, 3]. Это приводит к выводу на печать 1, затем рекурсивно вызывается write_a_list с аргументом в виде хвоста списка:

```
my::write_a_list([2, 3]).
 /* Это вызов write_a_list(T). */
```

Этот второй вызов опять сопоставляется со вторым клаузом, где, на этот раз H=2 и T=[3], поэтому выводится 2 и опять рекурсивно вызывается write_a_list:

```
my::write_a_list([3]).
```

С каким клаузом теперь такой вызов сопоставлятся? Напомним, что, хотя список [3] имеет всего один элемент, у него есть голова и хвост - голова есть 3, а хвост есть []. Таким образом, этот вызов опять сопоставляется со вторым клаузом с H=3 и T=[]. Теперь выводится 3 и вызывается рекурсивно write_a_list:

```
my::write_a_list([]).
```

Теперь становится понятно для чего нужен первый клауз. Второй клауз не может быть сопоставлен с таким вызовом, поскольку [] не может быть разделен на голову и хвост. Если бы первого клазуа здесь не было бы, то выполнение goal оказалось бы неуспешным. Но, поскольку он есть, то первый клауз сопоставляется с вызовом и выполнение goal успешно завершается и нечего более не делается.

Подсчет элементов в Списке

Рассмотрим теперь, как подсчитать число элементов в списке, или какова длина списка? Логично определить:

- длина пустого списка [] есть 0;
- длина любого другого списка есть 1 плюс длина его хвоста.

Можно ли это запрограммировать? На Прологе это очень просто. Всего два клауза:

```
class my
predicates
  length_of : (A*, integer) procedure(i,o).
end class

implement my
clauses
  length_of([], 0).
  length_of([_|T], L):-
    length_of(T, TailLength),
    L = TailLength + 1.
end implement

goal
  console::init(),
  my::length_of([1, 2, 3], L),
  stdio::write(L).
```

Посмотрите прежде всего на второй клауз. Строго говоря, [_|T] сопоставляется с любым непустым списком, связывая T с хвостом списка. Значение головы неважно, если она есть, она может быть учтена как один элемент.

Тогда вызов:

```
my::length_of([1, 2, 3], L)
```

сопоставляется со вторым клаузом, с T=[2, 3]. Следующим шагом является вычисление длины хвоста T. Когда это сделано (не имеет значение, как), TailLength получит значение 2, и компьютер теперь может добавить 1 к ней и связать L со значением 3. Как выполняется этот промежуточный шаг? Надо найти длину списка [2, 3], путем удовлетворения цели

```
my::length_of([2, 3], TailLength)
```

Другими словами, length_of вызывает себя рекурсивно. Этот вызов сопоставляется со вторым клаузом, связывая [3] и T в вызове клаузы и TailLength с L в клаузе.

Подчеркиваем, TailLength в вызове никак не пересекается с TailLength в клаузе, поскольку каждый рекурсивный вызов клауза имеет собственный набор переменных.

Итак, теперь задача - найти длину списка [3], которая есть 1, и мы добавляем 1 к этому значению, чтобы получить длину списка [2, 3], что будет 2. Ну и хорошо!.

Аналогично, length_of вызывает себя рекурсивно опять для получения длины списка [3]. Хвост [3] есть [], поэтому T связывается с [], и задача теперь - получение длины списка [] и добавление к ней 1, что дает длину списка [3].

Теперь все просто. Цель:

```
my::length_of([], TailLength)
```

сопоставляется с первым клаузом, связывая TailLength с 0. Поэтому теперь компьютер может добавить 1 к нему, получая длину списка [3], и возвращаясь теперь в вызывавший клауз. Это, в свою очередь, опять добавляет 1, давая длину списка [2, 3], и возвращается в клауз, который его вызывал; этот первоначальный клауз добавит снова 1, давая длину списка [1, 2, 3].

В следующей короткой иллюстрации сводим воедино все вызовы. Используется прием подстрочника для того, чтобы показать, что аналогично называемые переменные в разных клаузах или различные вызовы того же самого клауза - одно и то же.

```
my::length_of([1, 2, 3], L1).
my::length_of([2, 3], L2).
my::length_of([3], L3).
my::length_of([], 0).
L3 =  0+1 = 1.
L2 = L3+1 = 2.
L1 = L2+1 = 3.
```

Обратите внимание, что Вам не нужно каждый раз создавать такого рода предикаты самостоятельно, Вы можете использовать готовый предикат **list::length** из PFC.

Хвостовая рекурсия

Вы, очевидно, заметили, что length_of не является (и не может быть) предикатом с хвостовой рекурсией, поскольку рекурсивный вызов не является последним шагом в его клаузе. Возможно ли создать предикат,

определяющий длину, так, чтобы он был предикатом с хвостовой рекурсией? Да, но это потребует некоторых усилий.

Проблема с предикатом length_of в том, что длину списка нельзя вычислить до тех пор, пока не вычислена длина его хвоста. Но из этой ситуации есть выход. Нам потребуется предикат, вычисляющий длину списка, с тремя аргументами.

1. Один из них - это список, от которого компьютер будет откусывать по одному элементу на каждом вызове до тех пор, пока этот список, как и прежде, не превратится в пустой список.
2. Второй - это свободный аргумент, который в конечном итоге вернет результат (длину).
3. Третий - это счетчик, значение которого начинается с нуля и увеличивается с каждым вызовом.

Когда список в конечном итоге станет пустым, мы проунифицируем счетчик с несвязанным результатом.

```
class my
predicates
  length_of : (A*, integer, integer) procedure(i,o,i).
end class

implement my
clauses
  length_of([], Result, Result).
  length_of([_|T], Result, Counter):-
    NewCounter = Counter + 1,
    length_of(T, Result, NewCounter).
end implement

goal
  console::init(),
  my::length_of([1, 2, 3], L, 0), /* Начинаем со счетчиком Counter = 0 */
  stdio::write(" L = ", L).
```

Эта версия предиката length_of более сложная и во многих смыслах менее логичная, чем предыдущая. Мы ее представили здесь главным образом для того, чтобы показать, что на практике вы можете часто построить алгоритм с хвостовой рекурсией для задач, которые на первый взгляд требуют рекурсии другого типа.

Модификация Списка

Иногда требуется создать другой список из заданного списка. Это делается путем просмотра списка, элемент за элементом, заменяя каждый элемент вычисленным значением. Например, как эта программа, которая добавляет 1 к каждому элементу исходного списка:

```
class my
predicates
  add1 : (integer*, integer*) procedure(i,o).
end class

implement my
clauses
  add1([], [])./* граничное условие */
  add1([Head|Tail],[Head1|Tail1]):- /* отделяем голову от остального списка*/
    Head1 = Head+1, /* добавляем 1 к элементу-голове */
    add1(Tail, Tail1)./* далаем это с остальной частью списка*/
end implement

goal
  console::init(),
  my::add1([1,2,3,4], NewList),
  stdio::write(NewList)).
```

На обычном языке это звучит так:

- добавление 1 ко всем элементам пустого списка порождаем пустой список,
- для добавления 1 ко всем элемента любого другого списка:
    - добавить 1 к голове и сделать эту голову головой результирующего списка, а затем
    - добавить 1 к каждому элемента хвоста и этот хвост сделать хвостом результата.

Загрузим программу и выполним такую цель

```
add1([1,2,3,4], NewList).
```

Цель вернет

```
NewList=[2,3,4,5]
1 Solution
```

Опять о хвостовой рекурсии

Является ли предикат add1 проедикатом с хвостовой рекурсией? Если у Вас есть опыт использования Lisp или Pascal, Вы могли бы подумать, что нет, поскольку Вы бы рассуждали так:

- делим список на Head и Tail;
- добавляем 1 к Head, получаем Head1;
- рекурсивно добавляя 1 ко всем элементам списка Tail, получаем Tail1;
- соединяем Head1 и Tail1, что дает результирующий список.

Это не похоже на хвостовую рекурсию, поскольку последний шаг - не рекурсивный вызов.

Однако, и это важно, – Это не то, что делает Пролог. В Visual Prolog add1 является предикатом с хвостовой рекурсией, поскольку выполняется в действительности следующим образом:

- связать голову и хвост исходного списка с Head и Tail, соответственно;
- связать голову и хвост результирующего списка с Head1 и Tail1, соответственно. (Head1 и Tail1 пока не получили значений);
- добавить 1 к Head, что дает Head1;
- рекурсивно добавить 1 ко всем элементам списка Tail, что дает Tail1.

Когда это сделано, Head1 и Tail1 уже являются головой и списком результата и отдельной операции по их соединению нет. Поэтому рекурсивный вызов и является последним шагом.

Снова модификация списков

Конечно, не всегда модификации подлежит каждый элемент. Посмотрим на программу, которая сканирует список чисел и копирует его, удаляя отрицательные числа:

```
class my
predicates
  discard_negatives : (integer*, integer*) procedure(i,o). /*удалить отрицательные*/
end class

implement my
clauses
  discard_negatives([], []).
  discard_negatives([H|T], ProcessedTail):-
    H < 0,
```

```
    !, /* Если H отрицательно, пропускаем его */
    discard_negatives(T, ProcessedTail).
  discard_negatives([H|T], [H|ProcessedTail]):-
    discard_negatives(T, ProcessedTail).
end implement

goal
  console::init(),
  my::discard_negatives ([2, -45, 3, 468], X),
  stdio::write(X).
```

Напрмер, цель

```
my::discard_negatives([2, -45, 3, 468], X)
```

дает

```
X=[2, 3, 468].
```

А вот - предикат который копирует элементы списка, добавляя для каждого элемента его дубликат:

```
doubletalk([], []).
doubletalk([H|T], [H, H|DoubledTail]) :-
  doubletalk(T, DoubledTail).
```

Принадлежность списку

Допустим, имеется список с именами John, Leonard, Eric и Frank и требуется, используя Visual Prolog, выяснить, принадлежит ли заданное имя этому списку. Другими словами, надо определить "отношение" между двумя аргументами: именем и списком имен. Это соответствует предикату

```
isMember : (name, name*).
  /* "name" принадлежит списку "name*" */
```

В программе e01.pro первый клауз исследует голову списка. Если голова списка совпадает с искомым именем, то можно сделать заключение, что Name принадлежит списку. Поскольку хвост списка нас не интересует, то это мы представляем анонимной переменной. Благодаря первому клаузу, цель

```
my::isMember("john", ["john", "leonard", "eric", "frank"])
```

удовлетворена.

```
/* Программа e01.pro */
class my
predicates
  isMember : (A, A*) determ.
end class
```

```
implement my
clauses
  isMember(Name, [Name|_]) :-
    !.
  isMember(Name, [_|Tail]):-
    isMember(Name,Tail).
end implement

goal
  console::init(),
  my::isMember("john", ["john", "leonard", "eric", "frank"]),
  !,
  stdio::write("Success")
  ;
  stdio::write("No solution").
```

Если голова списка не есть Name, то надо исследовать, не содержится ли Name в хвосте списка.

На обычном языке:

Name принадлежит списку, если Name является первым элементом списка, или Name принадлежит списку, если Name принадлежит хвосту.

Второй клауз предиката isMember относится к этому отношению. Таким образом на Visual Prolog:

```
isMember(Name, [_|Tail]) :-
    isMember(Name, Tail).
```

Добавление списка к другому списку: декларативное и рекурсивное решения

Рассмотренный предикат member программы e01.pro работает в двух направлениях. Вернемся к его клаузам:

```
member(Name, [Name|_]).
member(Name, [_|Tail]) :-
  member(Name, Tail).
```

На эти клаузы можно смотреть с двух различных точек зрения: декларативной и процедурной.

1. С декларативной точки зрения, клаузы выражают:
    - Name (Имя) принадлежит списку, если голова списка есть Name
    - иначе Name принадлежит списку, если оно (Имя) принадлежит хвосту.

2. С процедурной точки зрения, эти же два клауза могут быть интерпретированы так:

   Чтобы найти элемент списка

- найдите его голову;
- иначе найдите элемент хвоста списка.

Эти две точки зрения соответствуют целям

```
member(2, [1, 2, 3, 4]).
```

и

```
member(X, [1, 2, 3, 4]).
```

В результате первый вызов поручает Visual Prolog(y) проверить, истино ли нечто (принадлежность числа 2 списку [1,2,3,4]). Второй вызов поручает Visual Prolog(y) найти все члены списка [1,2,3,4]. Не смущайтесь этим. Предикат member является одним и тем же, но на его поведение можно смотреть под разными углами.

Рекурсия с процедуральной точки зрения

Прелесть Пролога заключается в том, что часто, когда мы конструируем клаузы для предиката, будучи на одной точке зрения, они будут работать и при взгляде с другой точки зрения. Чтобы обнаружить эту дуальность, мы приведем пример предиката для добавления (append) одного списка к другому. Мы определяем предикат append с тремя аргументами:

```
append(List1, List2, List3).
```

Этот предикат интегрирует списки List1 и List2 в форму списка List3 так, что список List2 дописывается в конце списка List1. То есть содержательно - осуществляется добавление списка List2 к списку LIst1. Опять мы используем рекурсию (на этот раз с процедуральной точки зрения).

Если список List1 пустой, результатом добавления списка List1 к списку List2 будет тот же самый List2. Запишем это на Прологе:

```
append([], List2, List2).
```

Если список List1 не пустой, то можно преобразовать списки List1 и List2 к форме списка List3, сделав голову списка List1 головой списка List3. В

приведенном коде переменная H используется в качестве головы как списка List1, так и списка List3. хвост списка List3 есть список L3, который составлен из остатка списка List1 (а именно, L1) и всего списка List2. Опять выразим это на Прологе:

```
append([H|L1], List2, [H|L3]) :-
  append(L1, List2, L3).
```

Предикат append работает следующим образом: пока список List1 не пустой, рекурсивное правило дописывает один элемент каждый раз к списку List3. Когда список List1 становится пустым, первый клауз clause обеспечивает дописывание списка List2 в конец списка List3.

Варианты использования одного предиката

Подходя к предикату append с декларативной точки зрения, мы определили его как отношение между тремя списками. Это отношение справедливо также, если списки List1 и List3 известны, а List2 - нет. Более того, это также работает, если только List3 известен. Например, для того, чтобы выяснить, какие два списка могли бы быть соединены для получения известного списка, можно использовать вызов в форме

```
append(L1, L2, [1, 2, 4]).
```

С таким целевым вызовом, Visual Prolog найдет следующие решения:

```
L1=[], L2=[1,2,4]
L1=[1], L2=[2,4]
L1=[1,2], L2=[4]
L1=[1,2,4], L2=[]
4 Solutions
```

Можно использовать предикат append для нахождения списка, который следовало бы добавить к списку [3,4] для получения списка[1,2,3,4]. Попробуем такой вызов

```
append(L1, [3,4], [1,2,3,4]).
```

Visual Prolog находит решение

```
L1=[1,2].
```

Предикат append определяет отношение между входным набором (input set) и выходным набором (output set) таким образом, что отношение применимо в обе стороны. При таком отношении возникает вопрос

Что является выходным набором для заданного входного? или Какой входной набор соответствует заданному выходному?

Статус аргументов данного вызова предиката известен как поток (или шаблон) ввода-вывода. Аргумент, который связан или наследуется в момент вызова является входным аргументом и обозначается как (i). Свободный аргумент является выходным аргументом и обозначается как (o).

Предикат append обладает свойством поддерживать любой шаблон ввода-вывода, какой требуется. Однако не все предикаты имеют возможность вызова с различными шаблонами ввода-вывода. Когда клауз Пролога способен поддерживать множество шаблонов ввода-вывода, он называется инверсным клаузом. Целый набор предикатов для обработки списков содержится в классе **list.**

Задания.

Решить задачу о чайной церемонии.

Дан целочисленный список. Разделить на два списка, четных и нечетных чисел.

Дан целочисленный список. Разделить на два списка, простых и непростых чисел.

Дан целочисленный список. Разделить на два списка, на первую и вторую половины.

Дан целочисленный список. Разделить на три списка: кратные 5, кратные 8 и другие числа.

Даны два целочисленных списка. Построить список их пересечения.

Даны два целочисленных списка. Построить список их объединения.

Даны два целочисленных списка (все числа двухзначные). Построить список их дополнения.

Практическая работа №9

«Нечеткие знания»

Цель работы: закрепить знания по вопросам представления знаний на основе нечетких знаний и вывода в них.

Выясните, как задается нечеткое множество и чем нечеткое множество отличается от четкого. Обратите внимание на обозначения в случае непрерывного и дискретного множества.

Изучите операции на нечетких множествах. Дайте графическую интерпретацию указанных операций.

Выясните, как задается нечеткое отношение. Изучите задание отношений в случае конечных множеств с помощью матрицы отношений и взвешенного графа. Обратите внимание на определение операции свертки max-min двух нечетких множеств.

Разберитесь с методом приближенных рассуждений, в котором посылки являются нечеткими понятиями. Уясните, как получаются нечеткие отношения из нечеткого условного высказывания. Сравните методы нечетких рассуждений, использующие перечисленные типы нечетких отношений в правилах обобщенного модус поненс.

Пусть имеются следующие посылки:

x – не очень маленькое;

если x – маленькое, то y – большое, иначе y – маленькое.

Найти значения y’. Множество U = 1+2+3.

маленькое = 1/1+ 0.4/2; большое = 0.5/2+ 1/3.

Квантификатор очень может интерпретироваться с помощью опе-рации концентрации, т.е. возведения в квадрат: очень $\mathbf{x} = \int \mu^2_X(u)/u$.

Квантификатор не может интерпретироваться с помощью операции отрицания: не $\mathbf{x} = \int (1-\mu_X(u))/u$.

Тогда термин очень маленькое = 1/1+ 0.16/2, а не очень маленькое = 0.84/2+ 1/3.

Отношение для максиминного правила:

$$Rm'=(A \times B) \cup (\sim A \times C) = \int_{U\times V} (\mu_A(u) \wedge \mu_B(v)) \vee ((1-\mu_A(u)) \wedge \mu_C(v))/(u,v)$$

здесь A = маленькое, B = большое, C = маленькое.

Пример вычисления значений элементов матрицы Rm':

$(u_1, v_1) = (1 \wedge 0) \vee (0 \wedge 1) = 0$, $(u_2, v_1) = (0.4 \wedge 0) \vee (0.6 \wedge 1) = 0.6$,
$(u_1, v_2) = (1 \wedge 0.5) \vee (0 \wedge 0.4) = 0.5$, $(u_2, v_2) = (0.4 \wedge 0.5) \vee (0.6 \wedge 0.4) = 0.4$,
$(u_1, v_3) = (1 \wedge 1) \vee (0 \wedge 0) = 1$, $(u_2, v_3) = (0.4 \wedge 1) \vee (0.6 \wedge 0) = 0.4$,
$(u_3, v_1) = (0 \wedge 0) \vee (1 \wedge 1) = 1$,
$(u_3, v_2) = (0 \wedge 0.5) \vee (1 \wedge 0.4) = 0.4$,
$(u_3, v_3) = (0 \wedge 1) \vee (1 \wedge 0) = 0$,

$$Rm' = \begin{vmatrix} 0 & 0.5 & 1 \\ 0.6 & 0.4 & 0.4 \\ 1 & 0.4 & 0 \end{vmatrix}$$

Тогда значение y' может быть определено следующим образом:

$$\textbf{\textit{y'}} = \textbf{\textit{не очень маленькое}} \bullet Rm' = \begin{vmatrix} 0 & 0.84 & 1 \end{vmatrix} \bullet \begin{vmatrix} 0 & 0.5 & 1 \\ 0.6 & 0.4 & 0.4 \\ 1 & 0.4 & 0 \end{vmatrix} = \begin{vmatrix} 1 & 0.4 & 0.4 \end{vmatrix},$$

т.е. y' = 1/1+ 0.4/2 + 0.4/3, что может быть интерпретировано (с некоторой натяжкой) как довольно-таки маленькое.

Далее рассмотрим для указанных выше посылок арифметическое правило Ra':

$$Ra' = (\sim A \times V + U \times B) \cap (A \times V + U \times C) =$$

$$\int_{U\times V} 1 \wedge (1-\mu_A(u)+\mu_B(v)) \wedge ((\mu_A(u)+\mu_C(v))/(u, v) = \begin{vmatrix} 0 & 0.5 & 1 \\ 0.6 & 0.8 & 0.4 \\ 1 & 0.4 & 0 \end{vmatrix}$$

Тогда значение y' при использовании арифметического правила может быть определено следующим образом

$$\textbf{\textit{y'}} = \textbf{\textit{не очень маленькое}} \bullet Ra' = \begin{vmatrix} 0 & 0.84 & 1 \end{vmatrix} \bullet \begin{vmatrix} 0 & 0.5 & 1 \\ 0.6 & 0.8 & 0.4 \\ 1 & 0.4 & 0 \end{vmatrix} = \begin{vmatrix} 1 & 0.8 & 0.4 \end{vmatrix},$$

т.е. y' = 1/1+ 0.8/2 + 0.4/3.

Вывод с использованием размытого бинарного правила приведен ниже:

$$Rb' = (\sim A \times V \cup U \times B) \cap (A \times V \cup U \times C) = \int_{U\times V} (1-\mu_A(u) \vee \mu_B(v)) \wedge ((\mu_A(u) \vee \mu_C(v))/(u, v)$$

=

$$\begin{vmatrix} 0 & 0.5 & 1 \\ 0.6 & 0.4 & 0.4 \\ 1 & 0.4 & 0 \end{vmatrix}.$$

Тогда значение y', может быть определено следующим образом

$$\textbf{\textit{y'}} = \textbf{\textit{не очень маленькое}} \bullet \text{Rb}' = \begin{vmatrix} 0 & 0.84 & 1 \end{vmatrix} \bullet \begin{vmatrix} 0 & 0.5 & 1 \\ 0.4 & 0.4 & 0.4 \\ 1 & 0.4 & 0 \end{vmatrix} = \begin{vmatrix} 1 & 0.4 & 0.4 \end{vmatrix},$$

т.е. y' = 1/1+ 0.4/2 + 0.4/3, что может быть также интерпретировано как довольно таки маленькое.

Последний нечеткий вывод проведем с использованием правила Танака-Мидзумото

$$\text{Rgg}' = (A \times V \Rightarrow U \times B) \cap (\sim A \times V \Rightarrow U \times C) = \int_{U \times V} (\mu_A(u) \to \mu_B(v)) \wedge ((1 - \mu_A(u)) \to \mu_C(v))/(u,v),$$

где $\mu_A(u) \to \mu_B(v) = \begin{cases} 1, \text{если } \mu_A \le \mu_B \\ \mu_B, \text{если } \mu_A > \mu_B \end{cases}$

$$\text{Rgg}' = \begin{vmatrix} 0 & 0.5 & 1 \\ 0 & 0.4 & 0 \\ 1 & 0.4 & 0 \end{vmatrix}$$

$$\textbf{\textit{y'}} = \textbf{\textit{не очень маленькое}} \bullet \text{Rgg}' = \begin{vmatrix} 0 & 0.84 & 1 \end{vmatrix} \bullet \begin{vmatrix} 0 & 0.5 & 1 \\ 0 & 0.4 & 0 \\ 1 & 0.4 & 0 \end{vmatrix} = \begin{vmatrix} 1 & 0.4 & 0 \end{vmatrix},$$

т.е. y' = 1/1+ 0.4/2 + 0/3, что интерпретируется как маленькое.

Задание 3. Для всех десяти вариантов пусть имеется следующее правило:

если x – маленькое, то y – большое, иначе y – маленькое.

Вторая посылка и значения переменных маленькое, большое приведено в вариантах;

U = 1+2+3+4.

Найти значения y, используя последовательно все четыре правила нечеткого вывода (максиминное, арифметическое, размытое бинарное и правило Танака-Мидзумото). Сравните результаты, насколько сильно они отличаются от ожидаемых.

Вариант 1. Вторая посылка: x – не большое. Значения переменных маленькое = 1/1+ 0.8/2+ 0.2/3, большое = 0.2/2+ 0.7/3+ 1/4.

Вариант 2. Вторая посылка: x – большое. Значения переменных маленькое = 1/1+ 0.8/2+ 0.2/3, большое = 0.2/2+ 0.75/3+ 1/4.

Вариант 3. Вторая посылка: x – очень большое. Значения переменных маленькое = 1/1+ 0.7/2+ 0.2/3, большое = 0.25/2+ 0.75/3+ 1/4.

Вариант 4. Вторая посылка: x – не маленькое. Значения переменных маленькое = 1/1+ 0.8/2+ 0.3/3, большое = 0.2/2+ 0.9/3+ 1/4.

Вариант 5. Вторая посылка: x – очень маленькое. Значения переменных маленькое = 1/1+ 0.7/2+ 0.1/3, большое = 0.2/2+ 0.8/3+ 1/4.

Вариант 6. Вторая посылка: x – маленькое. Значения переменных маленькое = 1/1+ 0.75/2+ 0.1/3, большое = 0.25/2+ 0.7/3+ 1/4.

Вариант 7. Вторая посылка: x – не маленькое. Значения переменных маленькое = 1/1+ 0.75/2+ 0.1/3, большое = 0.25/2+ 0.7/3+ 1/4.

Вариант 8. Вторая посылка: x – очень маленькое. Значения переменных маленькое = 1/1+ 0.7/2+ 0.1/3, большое = 0.2/2+ 0.8/3+ 1/4.

Вариант 9. Вторая посылка: x – не очень большое. Значения переменных маленькое = 1/1+ 0.7/2+ 0.2/3, большое = 0.25/2+ 0.75/3+ 1/4.

Вариант 10. Вторая посылка: x – очень большое. Значения переменных маленькое = 1/1+ 0.7/2+ 0.2/3, большое = 0.2/2+ 0.8/3+ 1/4.

Практическая работа №10

«Составные списки»

Цель работы: освоить основные приемы работы со списками в языке Пролог.

Составные Домены и Списки

(Перенаправлено с Составные Домены и Списки)

В этом руководстве мы представим составные домены (иногда называемые алгебраическими структурами данных). Составные домены используются для обработки наборов данных как единого целого. Списки являются примером составных доменов. Они используются настолько часто, что получили даже специальную синтаксическую окраску.

Составные домены и списки создаются с использованием встроенных (built-in) и других составных или списковых доменов. Справочная система Visual Prolog (Help) объясняет встроенные домены:

- integer – целые;
- real – вещественные;
- string – строковые;
- symbol –символьные;
- char – знаковые;
- sring8 - строковые 8-разрядные;
- pointer – указатели;
- binary - двоичные (бинарные);
- boolean – булевские;
- object – объекты.

Составные домены и Функторы

Составные домены позволяют рассматривать наборы данных как единое целое и при этом мы можете рассматривать их и раздельно. Рассмотрим, например, дату "October 15, 2003". Она состоит из трех единиц

информации – месяца, дня и года – но было бы полезно рассматривать дату как единое целое в виде древовидной структуры:

```
          DATE
           /|\
          / | \
         /  |  \
        /   |   \
  October 15  2003
```

Это можно сделать путем объявления домена **date_cmp**, содержащего данные **date**:

```
domains
    date_cmp = date(string Month, unsigned Day, unsigned Year).
```

и затем писать просто, то есть

```
D = date("October", 15, 2003),
```

Это выглядит как факт Пролога, но это не так – это всего лишь значение, которое можно обрабатывать почти также, как строки или числа. Такая структура начинается с имени, обычно называемым функтор (**functor**) (в данном случае - **date)**, за которым следуют три аргумента.

Обращаем Ваше внимание на то, что функтор в Visual Prolog не имеет ничего общего с функциями в языках программирования. Функтор не вызывает никаких вычислений, это всего лишь имя, которое идентифицирует составную величину и объединяет свои аргументы.

Аргументы составной величины могут быть, в свою очередь, составными. К примеру, Вы можете думать о чьем-либо дне рождения как о структуре данных, так, как показано:

```
              birthday
               /    \
              /      \
             /        \
            /          \
           /            \
          /              \
    person               date
     /  \                / | \
    /    \              /  |  \
"Per" "Schultze"  "Apr" 14 1960
```

На Прологе это может быть записано как:

```
birthday(person("Per", "Schultze"), date("Apr",14,1960)).
```

В этом примере видны две подчасти составного значения birthday: аргумент person("Per", "Schultze") и аргумент date("Apr", 14, 1960). Функторами этих величин являются person и date.

Унификация Составных Доменов

Значения составных доменов могут унифицироваться либо с одиночными переменными, либо с составными значениями, которые сопоставимы с ними (здесь возможны переменные, как части внутренней структуры). Это значин, что Вы можете использовать составные величины для передачи в виде целого набора данных, а затем разбирать их по частям путем операции унификации. К примеру,

```
date("April", 14, 1960)
```

сопоставляется с переменной X и связывает переменную X с date("April",14,1960). Кроме того,

```
date("April", 14, 1960)
```

сопоставляется с date(Mo,Da,Yr) и связывает Mo с "April", Da с 14 и Yr с 1960.

Использование Знака Равенства для Унификации Составных Доменов

Visual Prolog выполняет унификацию в двух случаях. Первый - это когда вызов сопоставляется с головой клаузы. Второй - это при использовании знака равно (=), который в действительности является инфиксной формой предиката (предикат, который находится между своими аргументами, вместо того, чтобы быть перед ними).

Visual Prolog осуществляет необходимые связывания для унификации значений по обе стороны от знака равно. Это полезно для выделения значений аргументов в составных величинах. Например, следующий код проверяет имеют ли два человека одинаковы фамилии, после чего второй человек получает тот же адрес, что и первый.

```
class my
  domains
    person = person(name, address).
    name = name(string First, string Last).
    address = addr(street, string City, string State).
    street = street(integer Number, string Street_name).
```

```
    predicates
        run :().
end class

implement my
    clauses
        run():-
            console::init(),
            P1 =  person(name("Jim", "Smith"),addr(street(5, "1st st"), "igo", "CA")),
            P1 = person(name(_, "Smith"), Address),
            P2 = person(name("Jane", "Smith"), Address),
            !,
            stdio::write("P1 = ", P1, "\n"),
            stdio::write("P2 = ", P2, "\n")
            ;
            stdio::write("No solution").
end implement

goal
    my::run.
```

Структура Данных Как Единое Целое

Составные данные могут рассматриваться как одиночные значения в клаузах Пролога, что значительно упрощает программирование. Рассмотрим следующий факт

```
owns("John", book("From Here to Eternity", "James Jones")).
```

в котором утверждается, что John владеет книгой "From Here to Eternity", написанной автором "James Jones". В то же время можно написать

```
owns("John", horse("Blacky")).
```

что может интерпретироваться как John владеет лошадью с именем Blacky.

Составными значениями в этих двух примерах являются

```
book("From Here to Eternity", "James Jones")).
```

и

```
horse("Blacky").
```

Если бы вместо этого было бы записаны два факта:

```
owns("John", "From Here to Eternity").
owns("John", "Blacky").
```

то нельзя было бы решить является ли Blacky названием книги или имнем лошади. С другой стороны, можно использовать первый компонент структуры – функтор, для обозначения различия между различными типами

данных. В этом примере использовались, соответственно функторы book и horse для того, чтобы показать различие между данными.

Помните: Составные данные состоят из функтора и аргументов, принадлежащих функтору, как показано ниже:

```
functor(argument1, argument2, ..., argumentN)
```

Пример Использования Составных Доменов

Важным свойством составных доменов является возможность легкой передачи сгруппированных данных с помощью одного аргумента. Рассмотрим случай поддержки базы данных телефонных номеров. В эту базу данных мы хотим включать также дни рождения своих друзей и членов их семей. Ниже приведен фрагмент кода, который можно было бы написать для этого:

```
predicates
    phone_list :(string First, string Last, string Phone, string Month, intege Day, integer
Year) determ.

clauses
    phone_list("Ed", "Willis", "422-0208", "aug", 3, 1955).
    phone_list("Chris", "Grahm", "433-9906", "may", 12, 1962).
```

Анализируя данные, замечаем, что факт phone_list имеет шесть аргументов; пять из них могут разбиты на два составных домена:

```
          person                 birthday
           /   \                 /  |   \
          /     \               /   |    \
First Name   Last Name     Month Day Year
```

Было бы более полезно представить эти факты в виде составных доменов. Отступив на шаг, видим, что персона (person) является отношением, а имя (First Name) и фамилия (Last Name) являются его аргументами. Кроме того, день рождения является также отношением с тремя аргументами: месяц (month), день (day) и год (year). Представление этих отношений средствами Пролога аналогично уже упоминавшемуся

```
owns("John", "From Here to Eternity").
owns("John", "Blacky").
```

Теперь можно переписать нашу небольшую базу данных с использованием этих составных доменов в базе данных.

```
domains
    name = person(string First, string Last).
    birthday = b_date(string Month, integer Day, integer Year).

class predicates
    phone_list :(name, string Ph_num, birthday) determ.
clauses
    phone_list(person("Ed", "Willis"), "422-0208",b_date("aug", 3, 1955)).
    phone_list(person("Chris", "Grahm"), "433-9906",b_date("may", 12, 1962)).
```

В этой программе появились объявления двух составных **доменов (domains)**. Мы детальнее рассмотрим эти составные структыры данных позднее в этой главе. Пока же мы обратим внимание на преимущества использования таких составных доменов.

Предикат phone_list теперь содержит три аргумента вместо прежних шести. Иногда разбиение данных на составные структуры упрощает программу и облегчает обработку данных.

Теперь добавим некоторые правила в программу. Предположим, нам надо получить список людей, чьи дни рождения выпадают на текущий месяц. Ниже приведена программа, решающая эту задачу; эта программа использует стандартный предикат date для получения текущей даты из внутренних часов компьютера. Предикат date возвращает текущий год (year), месяц (month) и день (day) часов компьютера.

```
class my
domains
  name = person(string First, string Last).
  birthday = b_date(string Month, integer Day, integer Year).
predicates
  phone_list :(name, string Ph_num, birthday) multi(o,o,o).
  get_months_birthdays :().
  convert_month :(string Name, integer Num) determ(i,o).
  check_birthday_month :(integer Month_num, birthday) determ.
  write_person :(name).
end class

implement my

clauses
  get_months_birthdays() :-
    stdio::write("****** This Month's Birthday List *****\n"),
    stdio::write(" First Name\t\t Last Name\n"),
    stdio::write("*********************************\n"),
    CurTime = time::new(),
    CurTime:getDate(_, ThisMonth, _),
```

```
    phone_list(Person, _, Date),
      check_birthday_month(ThisMonth, Date),
      write_person(Person),
    fail.
  get_months_birthdays() :-
    stdio::write("\n Press any key to continue:\n"),
    _ = console::readChar().

clauses
  write_person(person(FirstName, LastName)) :-
    stdio::write("  ", FirstName, "\t\t ",  LastName, "\n").

clauses
  check_birthday_month(Mon, b_date(Month, _, _)) :-
    convert_month(Month, Month1),
    Mon = Month1.

clauses
  phone_list(person("Ed", "Willis"), "11-1111", b_date("Jan", 3, 1955)).
  phone_list(person("Benjamin", "Thomas"), "222-2222", b_date("Feb", 5, 1965)).
  phone_list(person("Ray", "William"), "333-3333", b_date("Mar", 3, 1955)).
  phone_list(person("Tomas", "Alfred"), "444-4444", b_date("Apr", 29, 1975)).
  phone_list(person("Chris", "Gralm"), "555-5555", b_date("May", 12, 1975)).
  phone_list(person("Dastin", "Robert"), "666-6666", b_date("Jun", 17, 1975)).
  phone_list(person("Anna", "Friend"), "777-7777", b_date("Jul", 2, 1975)).
  phone_list(person("Naomi", "Friend"), "888-8888", b_date("Aug", 10, 1975)).
  phone_list(person("Christina", "Lynn"), "999-9999", b_date("Sep", 25, 1975)).
  phone_list(person("Kathy", "Ann"), "110-1010", b_date("Oct", 20, 1975)).
  phone_list(person("Elizabeth", "Ann"), "110-1111", b_date("Nov", 9, 1975)).
  phone_list(person("Aaron", "Friend"), "110-1212", b_date("Dec", 31, 1975)).
  phone_list(person("Jenifer", "Faitlin"), "888-8888", b_date("Aug", 14, 1975)).

clauses
  convert_month("Jan", 1).
  convert_month("Feb", 2).
  convert_month("Mar", 3).
  convert_month("Apr", 4).
  convert_month("May", 5).
  convert_month("Jun", 6).
  convert_month("Jul", 7).
  convert_month("Aug", 8).
  convert_month("Sep", 9).
  convert_month("Oct", 10).
  convert_month("Nov", 11).
  convert_month("Dec", 12).
end implement

goal
  console::init(),
  my::get_months_birthdays().
```

Каким образом составные домены помогают несложно увидеть, просмотрев код. Основная часть обработки производится в предикате get_months_birthdays.

1. Сначала создается окно предикатом console::init()
2. После этого пишется заголовок в окне, помогающий интерпретировать результаты.
3. В предикате get_months_birthdays программа использует встроенный (built-in) предикат date, чтобы получить значение текущего месяца.
4. Далее все в программе направлено на поиск в базе данных и получения списка людей, родившихся в текущем месяце. Прежде всего надо найти первого человека в базе данных. Вызов phone_list(Person, _, Date) связывает имя и фамилию человка с переменной Person путем связывания целого функтора person с этой переменной Person. Одновремиенно связывается день рождения человека с переменной Date. Обратите внимание, что для этого требуется всего одна переменная для запоминания полного имени человека и всего одна переменная Date для запоминания дня рождения. В этом и заключается мощность составных доменов.
5. Теперь программа передает день рождения человека в виде одной переменной Date. Это происходит в следующей подцели, где программа передает значение текущего месяца (month), представленного целым числом, и день рождения (обрабатываемого человека) в предикат check_birthday_month.
6. Посмотрите внимательно, что происходит. Visual Prolog вызывает предикат check_birthday_month с двуми параметрами: первый параметр связан с целым числом, а второй - с термом birthday. В голове правила, определяющего предикат check_birthday_month, первый аргумент сопоставляется с переменной Mon. Второй аргумент, Date, сопоставляется с b_date(Month, _,_).

Поскольку нас интересует только месяц рождения человека, то на месте как дня, так и года рождения мы используем анонимную переменную.

7. Предикат check_birthday_month сначала преобразует строку с названием месяца в целочисленное значение. Как только это сделано, Visual Prolog может сравнивать значение текущего месяца со значением месяца рождения человека. Если такое сравнение положительно (успешно), то цель check_birthday_month считается успешной, и обработка продолжается. Если же сравнение неуспешно (fails) - обрабатываемый человек не родился в текущем месяце, то Visual Prolog начинает операцию отката для поиска другого возможного решения задачи.
8. Следующая обрабатываемая подцель - write_person. У обрабатываемого человека день рождения выпадает на текущий месяц, следовательно можно печатать его имя в отчете. После печати информации, клауза завершается предикатом неуспешности (fail), что вызывает откат.
9. Откат всегда распространяется вверх до ближайшего недетерминированного вызова и пытается передоказать вызов. В нашей программе последним недетерминированным вызовом является вызов phone_list. Именно здесь программа выбирает следующего человека для обработки. Если людей в списке (в базе данных) больше нет, то текущая клауза завершается неуспешно (fails); Visual Prolog теперь пытается доказать эту цель путем просмотра быза правил ниже. Поскольку есть еще одна клауза, определяющая get_months_birthdays, Visual Prolog пытается доказать обращение к get_months_birthdays путем доказательства подцелей этой другой части клаузы.

Декларирование Составных Доменов

В этом разделе мы покажем как определяются составные домены. После компиляции программы, содержащей следующие отношения:

```
owns("John", book("From Here to Eternity", "James Jones")).
```

и

```
owns("John", horse("Blacky")).
```

можно запрашивать систему с помощью следующей цели:

```
owns("John", X)
```

Переменная X может быть связана с различными типами данных - книгой, лошадью или возможно другим типом, который будет определен. Поскольку предикат owns теперь определён, больше его нельзя использовать в смысле старого определения:

```
owns :(string, string).
```

Второй аргумент больше не относится к домену string. Вместо этого необходимо сформулировать новое определение предиката, такое как, например

```
owns :(name, articles).
```

Вы можете теперь описать домен articles в секции domains как показано здесь:

```
domains
  articles =
    book(string Title, string Author);
    horse(string Name).
/* Articles - это книги или лошади */
```

Точка с запятой здесь читаются как or (или). В нашем случае возможны две альтернативы: либо книга может быть определена своим названием и автором, либо лошадь определяется своим именем. Домены Title, Author и Name - все являются стандартными доменами string.

Дополнительные альтернативы могут быть легко добавлены в декларацию домена. К примеру, articles мог бы включать лодку, дом, или банковский счёт. Для лодки, можно было бы использовать функтор без аргументов. С другой стороны, может возникнуть желание представлять баланс счета в качестве банковской записи. Тогда объявления домена articles расширяются до:

```
domains
  articles =
    book(string Title, string Author);
    horse(string Name);
    boat;
    bankbook(real Balance).
```

Далее приведена полная программа, которая показывает как составной домен articles может быть использован в определениях предиката owns.

```
class my
domains
  articles =
    book(string Title, string Author) ;
    horse(string Name) ;
    boat ;
    bankbook(real Balance).
predicates
  owns :(string Name, articles) nondeterm(i,o) determ(i,i).
end class

implement my
  clauses
    owns("John", book("A friend of the family", "Irwin Shaw")).
    owns("John", horse("Blacky")).
    owns("John", boat).
    owns("John", bankbook(1000)).
end implement

goal
  console::init(),
  my::owns("John", Thing),
    stdio::write("Thing: ", Thing, "\n"),
  fail
  ;
  stdio::write("The end.").
```

Загрузим теперь эту программу в IDE и запустим её. Visual Prolog теперь ответит:

```
Thing: book("A friend of the family","Irwin Shaw")
Thing: horse("Blacky")
Thing: boat()
Thing: bankbook(1000)
The end.
```

Итак: правила объявлений

Дадим теперь общее представление способа записи объявлений составных доменов:

```
domain =
  alternative1(D, D, ...);
  alternative2(D, D, ...);
  ...
```

Здесь alternative1 и alternative2 являются равновозможными (но различными) функторами. Обозначение (D, D, ...) представляет список имен,

которые объявлены в любом месте либо относятся к стандартным типам доменов(таким как string, integer, real и т.д.).

Примечания:

- альтернативы разделяются точкой с запятой;
- каждая альтернатива состоит из фуктора и, возможно, списка доменов для соответствующих аргументов;
- если функтор не имеет аргументов, его можно записывать как alternativeN или alternativeN( ).

Многоуровневые Составные Домены

Visual Prolog позволяет организовывать многоуровневые составные домены. Например, в

```
book("The Ugly Duckling", "Andersen").
```

вместо использования фамилии автора, можно было бы использовать новую структуру, детализирующую автора глубже, включая имя и фамилию автора. Дав название новому составному домену author, теперь можно поменять описание книги на

```
book("The Ugly Duckling", author("Hans Christian", "Andersen")).
```

В старом объявлении домена

```
book(string Title, string Author).
```

второй аргумент функтора book, представляющий автора может представлять только одиночное имя, но этого теперь недостаточно. Надо теперь объявить, что автор представляется составным доменом, образованным из имени и фамилии. Это делается декларацией домена:

```
author = author(string First_name, string Last_name).
```

что приводит к следующим объявлениям:

```
domains
   articles = book(string Title, author Author); ...
      /* First level */
   author = author(string First_name, string Last_name).
      /* Second level */
```

Используя составные домены таким образом на разных уровнях, часто бывает полезно рисовать дерево:

```
        book
        /  \
       /    \
   title    author
            /   \
           /     \
    First_name  Last_name
```

Одна декларация домена всегда описывает только один уровень дерева, а не все дерево. В частности, домен book не может быть объявлен такой декларацией:

```
/* Not allowed */
book = book(string Title, author(string First_name, string Last_name)).
```

Объявления Составных Смешанных Доменов

Давайте обсудим три различных объявления доменов, которые можно включать в программы. Эти декларации позволяют использовать предикаты, которые

- принимают аргументы более чем одного типа;
- принимают переменное число аргументов одного типа;
- принимают переменное число аргументов, каждый из которых может иметь более одного типа.

Аргументы множественных типов

Для того, чтобы предикат в Visual Prolog допускал использование аргументов различного типа, необходимо использовать функторные объявления. Следующий пример, клауза your_age clause будет воспринимать аргументы типа age, которые могут иметь тип string, real или integer.

```
domains
    age = i(integer); r(real); s(string).

class predicates
    your_age :(age).
clauses
    your_age(i(Age)) :- stdio::write(Age).
    your_age(r(Age)) :- stdio::write(Age).
    your_age(s(Age)) :- stdio::write(Age).
```

При этом Visual Prolog не допускает декларации вида:

```
/* Не допустимо. */
domains
    age = integer; real; string.
```

Списки

Предположим, мы обрабатываем данные о предметах, которые может преподавать учитель. Код может выглядеть так:

```
class predicates
    teacher :(string First_name, string Last_name, string Class) determ.

clauses
    teacher("Ed", "Willis", "english1").
    teacher("Ed", "Willis", "math1").
    teacher("Ed", "Willis", "history1").
    teacher("Mary", "Maker", "history2").
    teacher("Mary", "Maker", "math2").
    teacher("Chris", "Grahm", "geometry").
```

Здесь необходимо повторять имя преподавателя для каждого предмета, который он (она) может преподавать. Для каждого предмета необходимо добавлять такой факт в базу данных. Хотя в данной ситуации это не вызывает трудности, можно найти школу, в которой сотни предметов, и тогда поддержка такой структур данных становится утомительной. Тогда было бы удобно создать такой аргумент для предиката, который мог бы воспринимать один или несколько значений.

В Прологе это делает **список**. В следующем коде аргумент class объявлен имеющим **тип список**. Покажем, как список представляется в Прологе.

```
domains
    classes = string*.   /* объявляет списковый домен*/

class predicates
    teacher :(string First_name,  string Last_name,  classes Classes) determ.
clauses
    teacher("Ed", "Willis", ["english1", "math1", "history1"]).
    teacher("Mary", "Maker", ["history2", "math2"]).
    teacher("Chris", "Grahm", ["geometry"]).
```

В этом примере код более лаконичен и проще читается, чем код предыдущего примера. Обратите внимание на объявление домена. Звездочка (*) означает, домен classes является списком строк. Так же легко можно объявить список целых:

```
domains
    integer_list = integer*.
```

Объявив домен, теперь просто его использовать, поместив его в качестве аргумента в декларации предиката в секции предикатов. Посмотрим пример использования списка целых:

```
domains
    integer_list = integer*.

class predicates
    test_scores :
        (string First_name,
         string Last_name,
         integer_list Test_Scores)
         determ.
clauses
    test_scores("Lisa", "Lavender", [86, 91, 75]).
    test_scores("Libby", "Dazzner", [79, 75]).
    test_scores("Jeff", "Zheutlin", []).
```

Обратите внимание, в случае Jeff Zheutlin список не содержит никаких элементов.

Еще один пример показывает как можно использовать списки для представления семейного дерева.

```
domains
    tree_list = tree*.
    tree = tree(string Text, tree_list TreeList).

class predicates
    family :(tree) determ.
clauses
    family(tree("Grandmother",
           [tree("John",
              [tree("Eric", []),
               tree("Mark", []),
               tree("Leonard", []) ] ),
            tree("Ellen", [])
           ]
        )).
```

Практическая работа №11
«Классифицирующие системы»

Цель работы: создание классифицирующей системы на языке Пролог.

В общем виде задачу классификации можно сформулировать следующим образом:

1) имеется конечное множество объектов, которые можно описать определенным набором признаков,

2) каждый объект принадлежит одному классу из некоторого фиксированного множества,

3) в задаче классификации по заданному набору признаков необходимо определить, к какому классу принадлежит объект.

При создании классифицирующего приложения на языке Пролог необходимо создать классифицирующую экспертную систему. Выполнение работы состоит из следующих этапов:

- выбор предметной области. Четкое (письменное) формулирование цели создания системы;
- формальное описание предметной области. В качестве формализма для описания должны быть выбраны продукционные правила;
- представление предметной области на языке ПРОЛОГ;
- формулировка запросов и оформление интерфейса.

Примерный круг предметных областей:

- лекарственные растения (грибы, ягоды);
- покупка компьютера (автомобиля, квартиры);
- кулинария;
- породы домашних (диких) животных;
- починка телевизора (автомобиля, компьютера);
- лечение собаки (кошки, человека).

ТРЕБОВАНИЕ К БАЗЕ ЗНАНИЙ: количество правил должно быть не менее 30.

Например. Рассмотрим семь животных распространенных пород. Ниже приведены продукционные правила, задающие описание животных. Здесь биологический класс – это птицы или млекопитающие.

Правило 1.

ЕСЛИ животное имеет волосы,

ТО это животное млекопитающее.

Правило 2.

ЕСЛИ животное дает молоко,

ТО это животное млекопитающее.

Правило 3.

ЕСЛИ животное имеет перья,

ТО это животное птица.

Правило 4.

ЕСЛИ животное умеет летать

И несет яйца,

ТО это животное птица.

Правило 5.

ЕСЛИ животное – млекопитающее

И ест мясо,

ТО это хищник.

Правило 6.

ЕСЛИ животное – млекопитающее

И имеет острые зубы,

И имеет когти,

И имеет посаженные впереди глаза,

ТО это хищник.

Правило 7.

ЕСЛИ животное – млекопитающее

И имеет копыта,

ТО это копытное.

Правило 8.

ЕСЛИ животное – млекопитающее

И жует жвачку,

ТО это копытное.

Правило 9.

ЕСЛИ животное – хищник

И имеет рыжевато-коричневую окраску,

И имеет темные пятна,

ТО это леопард.

Правило 10.

ЕСЛИ животное – хищник

И имеет рыжевато-коричневую окраску,

И имеет черные полосы,

ТО это тигр.

Правило 11.

ЕСЛИ животное – копытное

И имеет длинные ноги,

И имеет длинную шею,

И имеет рыжевато-коричневую окраску,

И имеет темные пятна,

ТО это жираф.

Правило 12.

ЕСЛИ животное – копытное

И имеет белый цвет,

И имеет черные полосы,

ТО это зебра.

Правило 13.

ЕСЛИ животное – птица

И не умеет летать

И имеет длинные ноги,

И имеет длинную шею,

И имеет бело-черную окраску,

ТО это страус.

Правило 14.

ЕСЛИ животное – птица

И не умеет летать,

И умеет плавать,

И имеет бело-черную окраску,

ТО это пингвин.

Правило 15.

ЕСЛИ животное – птица

И умеет очень хорошо летать, ТО это альбатрос.

Работа системы распознавания сводится к генерации гипотезы о принадлежности животного к тому или иному классу и к попытке подтвердить эту гипотезу. В нашем случае генерируется первая гипотеза: «распознаваемое животное – это млекопитающее». Для подтверждения данной гипотезы необходимо, чтобы пользователь утвердительно ответил хотя бы на один из вопросов: «имеет ли животное волосы» или «дает ли животное молоко». Если положительный ответ получен уже на первый вопрос, то система генерирует следующую гипотезу «это млекопитающее – хищник». Если же положительный ответ не получен на первый вопрос, то система задает второй вопрос. Если на него получен положительный ответ, генерируется гипотеза «млекопитающее – хищник», если получен отрицательный ответ, генерируется гипотеза: «распознаваемое животное – птица». Процесс порождения гипотез и их проверки длится до тех пор, пока есть подходящие для этого правила. Описание таких правил приведено ниже:

rule(1,"животное","млекопитающее",[1]).

rule(2,"животное","млекопитающее",[2]).

```
rule(3,"животное","птица",[3]).
rule(4,"животное","птица",[4, 5]).
rule(5,"млекопитающее","хищник",[6]).
rule(6,"млекопитающее","хищник",[7, 8, 9]).
rule(7,"млекопитающее","копытное",[10]).
rule(8,"млекопитающее","копытное ",[11]).
rule(9,"хищник","леопард",[12, 13]).
rule(10,"хищник","тигр",[12, 14]).
rule(11,"копытное","жираф",[15, 16, 12, 13]).
rule(12,"копытное","зебра",[17, 14]).
rule(13,"птица","страус",[18, 15, 16, 19]).
rule(14,"птица","пингвин",[18, 20, 19]).
rule(15,"птица","альбатрос",[21]).
```

Первый аргумент в предикате rule – это номер правила, второй – род, третий – вид, четвертый – список вопросов, подтверждающий отношение род-вид.

Для того чтобы применить ту или иную продукцию, необходимо собрать факты, задав пользователю вопросы. Однако, прежде чем задавать вопрос, необходимо быть уверенным в том, что этот вопрос уже не был задан ранее при подтверждении других промежуточных гипотез. Информация о заданном вопросе и полученном на него ответе хранится в отношении fact(X, Y) динамической базы данных, где X – номер вопроса, Y – ответ на этот вопрос ("да", "нет"). Если вопрос был уже задан и на него получен положительный ответ, то вывод успешно продолжается, если же получен отрицательный ответ, то система сообщает о неуспехе. Множество задаваемых вопросов приведено ниже.

```
ask(X):- fact(X, "да"),!.
ask(X):- fact(X, "нет"),!,fail.
ask(1):- write("оно имеет волосы?"), !, complete(1).
ask(2):- write("оно дает молоко?"), !, complete(2).
```

```
ask(3):- write("оно имеет перья?"), !, complete(3).
ask(4):- write("оно умеет летать?"), !, complete(4).
ask(5):- write("оно несет яйца?"), !, complete(5).
ask(6):- write("оно ест мясо?"), !, complete(6).
ask(7):- write("оно имеет острые зубы?"), !, complete(7).
ask(8):- write("оно имеет когти?"), !, complete(8).
    ask(9):- write("оно имеет посаженные впереди глаза?"), !,
            complete(9).
ask(10):- write("оно имеет копыта?"), !, complete(10).
ask(11):- write("оно жует жвачку?"), !, complete(11).
ask(12):- write("оно имеет рыжевато-коричневую окраску?"),
        !, complete(12).
ask(13):- write("оно имеет темные пятна?"), !, complete(13).
    ask(14):- write("оно имеет черные полосы?"), !, complete(14).
ask(15):- write("оно имеет длинные ноги?"), !, complete(15).
ask(16):- write("оно имеет длинную шею?"), !, complete(16).
ask(17):- write("оно имеет белый цвет?"), !, complete(17).
ask(18):- write("оно не умеет летать?"), !, complete(18).
   ask(19):- write("оно имеет бело-черную окраску?"), !, complete(19).
ask(20):- write("оно умеет плавать?"), !, complete(20).
    ask(21):- write("оно умеет очень хорошо летать?"), !,
            complete(21).
```

Процедура recognition(X) занимает центральное место в программной реализации продукционной системы. Процедура состоит из трех предложений. В первом предложении генерируется гипотеза (rule(N, X, Y, Z)) и ищется ее подтверждение (discover(Z)); если гипотеза подтверждается, то выдается соответствующее сообщение, если выдвинутая гипотеза не подтверждается, то генерируется следующая. Второе предложение процедуры описывает ситуацию, когда пользователь на все вопросы ответил отрицательно и системе не удалось выдвинуть ни одной гипотезы. И

наконец, третье предложение задает успешное окончание работы, когда была подтверждена хотя бы одна гипотеза.

```
recognition(X):- rule(N, X, Y, Z), discover(Z), !,
        write("_____", X, " -  ", Y, " по правилу ", N), nl,
        recognition(Y).
recognition("животное"):- write("это животное мне неизвестно"),!.
recognition(_).

discover([]).
discover([X|Y]):- ask(X), discover(Y).

complete(X):- nl, read(Y), assert(fact(X, Y)), Y="да".
```

Рассмотрим пример работы системы.

```
?- retractall(_), recognition("животное").
"оно имеет волосы?"
да
_____животное – млекопитающее по правилу 1
"оно ест мясо?"
        нет
        "оно имеет острые зубы?"
        да
"оно имеет когти?"
да
"оно имеет посаженные впереди глаза?"
да
_____млекопитающее – хищник по правилу 6
"оно имеет рыжевато-коричневую окраску?"
да
"оно имеет темные пятна?"
```

нет

"оно имеет черные полосы?"

да

_____хищник – тигр по правилу 10

Пример показывает полное распознавание животного.

Важный вопрос построения продукционной системы – это разработка структуры продукционного правила. Некоторые рекомендации приведены ниже:

- конструируйте правила, опираясь на структуру, присущую предметной области;
- используйте минимально достаточное количество условий при определении продукционного правила;
- избегайте противоречащих продукционных правил.

Каждое продукционное правило может быть независимым от других. Модульность правил позволяет легко модифицировать продукционную систему. Модификация заключается в добавлении, изменении, удалении правил и не затрагивает существующих процедур. Число правил в системе ограничено размерами памяти компьютера. Продукционные правила можно поместить и в динамическую базу данных, тогда возможно хранение правил и во внешней памяти компьютера.

На самостоятельную работу выносятся вопросы создания графического интерфейса на языке Visual Prolog.

Практическая работа №12
«Вопросы создания экспертных систем»

Цель работы: закрепить знания и навыки при моделировании и создании экспертных систем.

При реализации данной практической работы должны быть рассмотрены следующие разделы дисциплины.

1. Формализация предметной области. Выбор модели знаний для заданной предметной области.
2. Определение состава разработчиков для разработки экспертной системы для заданной предметной области.
3. Моделирование экспертной системы. Выбор программного продукта.

Уясните, к какому классу относятся экспертные системы. Выясните, какие нововведения делают экспертную систему эффективной, ценной и имеют наибольшее потенциальное значение.

Изучите особенности архитектуры экспертной системы. Обратите внимание на компоненты, присущие только экспертным системам.

Процесс построения экспертных систем называют инженерией знаний. Разберитесь с тем, кто участвует в проектировании экспертных систем, выясните их функции и способы взаимодействия.

Выявите типичные задачи, выполняемые экспертными системами. Обратите внимание на характерные особенности, присущие этим задачам: большое пространство решений, условное рассуждение, применение зашумленных данных.

Изучите технологию проектирования и разработки экспертных систем. Выясните, какие задачи решаются на каждом из этапов разработки. Сравните технологию проектирования экспертных систем с технологией проектирования традиционных программных систем.

Выясните, что представляют собой инструментальные средства, облегчающие создание экспертных систем. Проведите сравнительный анализ средств, используемых для построения экспертных систем.

Выясните, чем вызвана важность объяснительного компонента в экспертных системах. Рассмотрите классификацию типов объяснения.

Рассмотрите методологические и технологические проблемы приобретения знаний. Изучите фазы и модели приобретения знаний.

Рассмотрите методы информатизации предприятий и их подразделений на основе Web- и CALS-технологий.

CALS (Continuous Acquisition and Life Cycle Support) – непрерывная информационная поддержка жизненного цикла продукции. Изначально была применена в 1980 – х годах в оборонном комплексе США как компьютерная поддержка поставок (Computer Aided Logistic Support). В дальнейшем распространилась на другие сферы экономики и на весь жизненный цикл продукта (от маркетинга до утилизации).

Разработка концепции CALS обусловлена развитием таких новых направлений науки и техники, как автоматизированное проектирование, управление производством, использование компьютеров для хранения и обработки информации, новые средства связи и другое. Каждое из этих направлений в отдельности внесло революционные изменения во все виды человеческой деятельности, однако их значительные возможности использовались недостаточно. Причиной стало то, что разработчики современных средств автоматизации формировали свои собственные модели, которые нередко оказывались несовместимыми у партнеров по производству и эксплуатации техники. Отчасти эта проблема решалась увязкой различных систем автоматизированного проектирования (САПР) в интегрированные системы путем физического объединения баз данных, однако логическая увязка при этом отсутствовала, что приводило к фрагментации информации, многократному дублированию данных, невозможности интеграции различных интегрированных автоматизированных систем управления

(ИАСУ). Решение проблемы следовало искать на пути информационных представлений и процессов, организации активного обмена согласованной информацией такого рода между партнерами. Так появилась концепция CALS. В отличие от автоматизированной системы управления производством АСУП и от ИАСУ CALS-технологии охватывают все стадии жизненного цикла продукции. Предмет CALS – технологии совместного использования и обмена информацией в процессах, выполняемых в течение жизненного цикла продукта. Базовыми принципами CALS являются:

1. Безбумажный обмен информацией.
2. Анализ и реинжиниринг бизнес-процессов.
3. Параллельный инжиниринг.
4. Системная организация постпроизводственных процессов жизненного цикла изделия.

Нормативную базу применения CALS технологий составляют различные международные стандарты (например ИСО 10303 – Система автоматизации производства и их интеграция). Преимущества использования CALS технологий:

1. Расширяются области деятельности предприятий за счет кооперации с другими предприятиями, обеспечиваемой стандартизации предоставления информации на разных стадиях и этапах жизненного цикла.
2. Повышается эффективность бизнес-процессов.
3. Повышается конкурентоспособность продукции.
4. Сокращаются затраты и трудоемкость процессов технической подготовки и освоения производства новых изделий.
5. Сокращаются календарные сроки вывода новых видов продукции на рынок.
6. Сокращается доля брака и затрат, связанных с внесением изменений в конструкцию.

7. Сокращаются затраты на эксплуатацию, обслуживание и ремонты изделий.

Задание.

Рассмотреть и формализовать все вопросы по построению экспертной системы для заданной, согласно индивидуального задания, предметной области. Провести обзор CALS стандартов при проектировании заданной системы.

Список литературы.

1. Образовательный стандарт вуза ОС ТУСУР 01-2013. Работы студенческие по направлениям подготовки и специальностям технического профиля. Общие требования и правила оформления. Введен приказом ректора от 03.12.2013 г. №14103. [Электронный ресурс]. URL: http://old.tusur.ru/export/sites/ru.tusur.new/ru/education/documents/inside/tech_01-2013_new.pdf (дата обращения 19.10.2017).
2. И. А. Ходашинский И. А., Методы искусственного интеллекта, базы знаний, экспертные системы : Учебное пособие / И. А. Ходашинский ; Министерство образования Российской Федерации, Томский государственный университет систем управления и радиоэлектроники, Кафедра автоматизации обработки информации. - Томск : ТУСУР, 2002. - 140 с.
3. Visual Prolog. Category:Tutorials. [Электронный ресурс]. URL: http://wiki.visual-prolog.com/index.php?title=Category:Tutorials (дата обращения 05.12.2017)
4. Ходашинский И. А., Язык ПРОЛОГ в примерах и задачах : учебное пособие для вузов /; Федеральное агентство по образованию, Томский государственный университет систем управления и радиоэлектроники. - Томск : ТУСУР, 2006. - 279 с.