Часть 5. СИСТЕМЫ ИСКУССТВЕННОГО ИНТЕЛЛЕКТА

20. ОСНОВЫ ИСКУССТВЕННОГО ИНТЕЛЛЕКТА

20.1. Понятия искусственного интеллекта

Искусственный интеллект (ИИ) **–** это научная дисциплина, возникшая в 50-х годах на стыке кибернетики, лингвистики, психологии и программирования. С самого начала исследования в области ИИ пошли по двум направлениям.

Первое (бионическое) – попытки смоделировать с помощью искусственных систем психофизиологическую деятельность человеческого мозга с целью создания искусственного разума.

Второе (прагматическое) – создание программ, позволяющих с использованием ЭВМ воспроизводить не саму мыслительную деятельность, а являющиеся ее результатами процессы. Здесь достигнуты важные результаты, имеющие практическую ценность. В дальнейшем речь будет идти об этом направлении.

Разработка интеллектуальных программ существенно отличается от обычного программирования и ведется путем построения системы искусственного интеллекта (СИИ). Если обычная программа может быть представлена в парадигме:

Программа=Алгоритм+Данные,

то для СИИ характерна другая парадигма:

СИИ=Знания + Стратегия обработки знаний.

Основным отличительным признаком СИИ является работа со знаниями. Если для обычных программ представление данных алгоритма определяется на уровне описания языка программирования, то для СИИ представление знаний выливается в проблему, связанную со многими вопросами: что такое знания, какие знания хранить в системе в виде *базы знаний* (БЗ), в каком виде и сколько, как их использовать, пополнять и т. д.

В отличие от данных, знания обладают следующими свойствами:

- *внутренней интерпретируемостью* – вместе с информацией в БЗ представлены информационные структуры, позволяющие не только хранить знания, но и использовать их;
- *структурированностью* – выполняется декомпозиция сложных объектов на более простые и установление связей между ними;
- *связанностью* – отражаются закономерности относительно фактов, процессов, явлений и причинно-следственные отношения между ними;
- *активностью* – знания предполагают целенаправленное использование информации, способность управлять информационными процессами по решению определенных задач.

Все эти свойства знаний в конечном итоге должны обеспечить возможность СИИ моделировать рассуждения человека при решении прикладных задач – со знаниями тесно связано понятие процедуры получения решений задач (стратегии обработки знаний). В системах обработки знаний такую процедуру называют *механизмом вывода, логическим выводом* или *машиной вывода*. Принципы построения механизма вывода в СИИ определяются способом представления знаний и видом моделируемых рассуждений.

Для организации взаимодействия с СИИ в ней должны быть средства общения с пользователем, т.е. *интерфейс.* Интерфейс обеспечивает работу с БЗ и механизмом вывода на языке достаточно высокого уровня, приближенном к профессиональному языку специалистов в той прикладной области, к которой относится СИИ. Кроме того, в функции интерфейса входит поддержка диалога пользователя с системой, что дает пользователю получать *объяснения* действий системы, участвовать в поиске решения задачи, пополнять и корректировать базу знаний. Таким образом, основными частями систем, основанных на знаниях, являются:

1. База знаний.
2. Механизм вывода.
3. Интерфейс с пользователем.

Каждая из этих частей может быть устроена по-разному в различных системах, отличия эти могут быть в деталях и в принципах. Однако для всех СИИ характерно *моделирование человеческих рассуждений*. СИИ создаются для того, чтобы овеществлять в рамках программно-технической системы знания и умения, которыми обладают люди, чтобы решать задачи, относящиеся к области творческой деятельности человека.

Знания, на которые опирается человек, решая ту или иную задачу, весьма разнородны. Это, прежде всего:

- понятийные знания (набор понятий и их взаимосвязи);
- конструктивные знания (знания о структуре и взаимодействии частей различных объектов);
- процедурные знания (методы, алгоритмы и программы решения различных задач);
- фактографические знания (количественные и качественные характеристики объектов, явлений и их элементов).

Особенность систем представления знаний заключается в том, что они моделируют деятельность человека, осуществляемую часто в неформальном виде. Модели представления знаний имеют дело с информацией, получаемой от экспертов, которая часто носит качественный и противоречивый характер. Для обработки с помощью ЭВМ такая информация должна быть приведена к однозначному формализованному виду. Методологией формализованного представления знаний является логика. Перейдем к рассмотрению основных понятий логики.

20.2. Введение в логику

Логика занимается изучением законов мышления, одной из главных ее задач является моделирование правильных человеческих рассуждений. Особый интерес к логике возник с появлением ЭВМ, с попытками научить машину рассуждать, т.е. делать логические заключения. Рассмотрим основные идеи, лежащие в основе логических рассуждений.

В логике выделяют следующие *формы мышления:* понятия, высказывания и рассуждения.

Понятие о предмете составляет совокупность мыслимых признаков предмета. Понятие выражается словом. Основными способами образования понятий являются:

- сравнение – установление сходства или различия в понятиях;
- анализ – мысленное расчленение целого на составные части:
- синтез – мысленное создание целого из некоторого числа составных частей (признаков, свойств, отношений);

- абстрагирование – мысленное выделение в понятии определенных признаков и отвлечение от других;
- обобщение – объединение различных объектов в однородные группы на основании присущих им общих признаков.

Всякое понятие обладает *содержанием* – совокупностью признаков предмета в данном понятии и объёмом – совокупностью объектов, входящих в данное понятие.

По содержанию понятия делятся на положительные и отрицательные (присутствуют или нет определенные признаки в понятии), безотносительные и относительные, сравнимые и несравнимые.

В логическом отношении друг с другом находятся только сравнимые понятия. Сравнимые понятия (имеющие общие признаки в содержании) бывают совместимыми (с совпадающим объемом понятий) и несовместимыми. Выделяют три вида отношений совместимости: равнозначность, пересечение, подчинение объемов. Существуют и три вида отношений несовместимости: соподчинение, противоположность, противоречие.

Связь между объемом и содержанием понятий выражается в *законе обратного отношения:* если два понятия, сравнимы в логическом смысле, и содержание первого из них больше, чем второго, тогда объем второго понятия больше объема первого. С этим законом связаны способы *обобщения* (переход от понятия с большим объемом и меньшим содержанием к понятию с меньшим объемом и большим содержанием) и *ограничения* (переход от понятия с меньшим объемом и большим содержанием к понятию с меньшим содержанием и большим объемом) понятий.

Одной из основных логических операций над объемом и содержанием понятий является *деление* понятия. При ее выполнении различают: делимое понятие, основание деления (признаки), члены деления (множество видовых понятий по отношению к рассматриваемому). Деление бывает: по *видоизменению* признаков, *дихотомическое* (деление понятия на два класса с противоречивыми признаками).

Практическое применение операции деления понятий – процедура классификации. Цель классификации – приведение знаний о предметной области в построенную систему, при этом основание деления должно отвечать цели классификации. Выбор классификационного признака – нетривиальная задача. Классификация, особенно при дихотомическом делении, принимает форму дерева (впервые использовалась сирийским логиком в четвертом веке Порфирием, называется по его имени ”дерево Порфирия”).

Другой фундаментальной логической операцией над понятиями является определение понятия. Эта операция позволяет строго закрепить за объектом, обозначенным с помощью определяемого понятия, содержание, выраженное в зафиксированных в определении признаках, свойствах и отношениях.

Определение бывает номинальное – раскрытие смысла употребления слова и реальное – определение понятий о предметах или явлениях, а не терминов, их обозначающих. Типы определений: через ближайший род и видовое отличие (это, по сути, поиск места понятия в некоторой явной или неявной форме классификации), способы задания определений: генетическое определение; операциональное определение.

Понятия являются исходным материалом для построения высказываний (суждений). С грамматической точки зрения, высказывания – это повествовательное предложение.

Сложные предложения строятся из выражений, обозначающих некоторые понятия, и логических связок. Слова и фразы “НЕ”, ”И”, ”ИЛИ”, ”ЕСЛИ... ТО”, ”ТОГДА И ТОЛЬКО ТОГДА”, ”СУЩЕСТВУЕТ”, ”ВСЕ” и некоторые другие называются *логическими связками* (операторами) и обозначают логические операции, с помощью которых из одних предложений строятся другие.

Предложения без логических связок являются *элементарными*, их нельзя расчленить на части, чтобы при этом каждая из частей была также предложением. Элементарные предложения называют также *высказываниями* (суждениями). В высказываниях содержится информация о предметах, явлениях, процессах и т.д.

Элементарное высказывание состоит из субъекта (логического подлежащего) – того, о чем идет речь в высказывании, и предиката (логического сказуемого) – того, что утверждается или отрицается в высказывании о субъекте.

Таким образом, ***высказывания*** – это форма мышления, в которой утверждается или отрицается логическая связь между понятиями, выступающими в качестве субъекта и предиката данного высказывания. Соответствие или несоответствие этой связи реальности делает высказывание (суждение) истинным или ложным.

Логическая связь между субъектом и предикатом высказывания выражается обычно в виде связки “ЕСТЬ” или “НЕ ЕСТЬ”, хотя в самом предложении эта связка может отсутствовать, а лишь подразумеваться. При этом субъект высказывания может выражаться не обязательно только подлежащим в предложении, также как и предикат не только сказуемым (это могут быть и другие члены предложения). Что считать в предложении субъектом, а что предикатом высказывания, определяется логическим ударением. Логическое ударение связано со смыслом, содержащимся в предложении для говорящего и для слушающего.

По форме высказывания делятся на *простые* (имеют логическую форму ”S есть P “ или ”S не есть P“ , где S – субъект, P – предикат) и *сложные* (грамматически выражаются сложными предложениями).

Простые высказывания позволяют выразить следующие типы высказываний:

- атрибутивные высказывания – выражают принадлежность или не принадлежность свойства объекту или классу объектов;
- высказывания об отношениях – говорят о наличии отношения между объектами;
- высказывания существования (экзистенциональные высказывания) – говорят о существовании или не существовании объекта или явления.

В общем случае простые высказывания можно рассматривать как атрибутивные, понимая под существованием и отношениями вид свойств субъектов высказывания.

По качеству простые высказывания делятся на утвердительные и отрицательные.

По количеству высказывания делятся на:

- единичные – субъектом является предмет, существующий в единственном числе;
- частные – в высказывании говорится о пересечении класса предметов, к которому относится субъект, с классом предметов, к которым относится предикат);
- общие – в высказывании говорится о включении или не включении всего класса предметов, к которому относится субъект высказывания, в класс предметов, к которому относится предикат.

Классы предметов, к которым относятся субъект и предикат высказывания, будем обозначать буквами S и P соответственно.

Высказывания, одновременно общие по количеству и утвердительные по качеству называются *общеутвердительными* и имеют форму “Всякий S есть P”, обозначается тип высказывания A.

Высказывания, общие по количеству и отрицательные по качеству, называются *общеотрицательными* и имеют форму “Всякий S не есть P”, обозначается тип высказывания E.

По данной аналогии выделяют *частноутвердительные* высказывания (“Некоторый S есть P”, обозначается тип как I) и *частноотрицательные* высказывания (“Некоторый S не есть P”, обозначается тип как O).

Различают также высказывания сравнимые – имеют одни и те же субъект и предикат и несравнимые – различны субъекты и предикаты суждений.

Третья форма мышления – *рассуждения.* Простейшей формой рассуждений является умозаключение – получение из одного или нескольких высказываний нового высказывания. Принято считать, что из высказываний A_1, A_2,..., A_n следует высказывание B, если B истинно, по крайней мере, всегда, когда истинны A_1, A_2,..., A_n. При этом исходные высказывания A_1, A_2,..., A_n, из которых делается логический вывод называются *посылками*, а новое высказывание B – *заключением, следствием.* Возможность вывода заключения из посылок обеспечивается логической связью между ними. Проверить правильность вывода из посылок можно логическими средствами без обращения к непосредственному опыту.

Правильные с позиций логики выводы формулируются в виде *правил логического следования* (*правил вывода*). Правила в логике обычно записываются в виде: A_1, A_2,...,$A_n \Rightarrow$B, здесь знак $\Rightarrow$ – знак логического следования; записываются правила также в виде дроби:

$$\frac{A_1, A_2,...,A_n}{B}$$

Таким образом, рассуждения – это процесс перехода от посылок к заключениям и далее от полученных заключений как новых посылок к новым заключениям. Выполняется этот процесс в виде элементарных актов, каждый из которых есть шаг вывода, на котором применяется соответствующее правило вывода и называется этот процесс обычно **логическим выводом**. Число посылок в выводе, как и число его шагов, может быть различным. Вывод за один шаг применения правила вывода называют непосредственным выводом.

Логическими законами называют ***нуль–посылочные выводы*** – высказывания, для выяснения истинности которых посылки не нужны. Наиболее важными являются законы: тождества, противоречия, исключенного третьего и достаточного основания.

Закон тождества: ”Объем и содержание всякого понятия должны быть зафиксированы и оставаться неизменными в течение всего процесса рассуждения”. Этот закон записывается с помощью формулы **A⇔A,** где ⇔ – знак эквивалентности, т. е. если два понятия тождественны, то они могут быть взаимозаменяемы в логическом контексте. Буквой **A** здесь обозначаем переменную для высказываний или высказывательную форму.

Закон противоречия: "Два противоречащих друг другу высказывания не могут быть одновременно истинными, по крайней мере, одно из них ложно". Формульная запись закона имеет вид: **¬(A ∧¬A)** – “неверно, что **A** и не **A**”.

Закон исключенного третьего (латинская формулировка этого закона – *Tertium non datur* –“Третьего не дано”) выражается формулой: **A∨¬A**, т.е. “истинно **A** или не **A**” или словесная формулировка – если два высказывания противоречат друг другу, то одно из них истинно, а другое ложно.

Закон достаточного основания предложен немецким философом Г.Лейбницем, он требует, чтобы ни одно утверждение не признавалось справедливым без достаточного основания. Закон в целом не выражается какой-либо формулой, его требования носят содержательный характер.

Двух посылочные выводы называют *силлогизмами* Аристотеля. Конкретные типы силлогизмов называют *модусами*. В модусе всегда в посылках присутствуют три понятия (большой, средний и малый термины). Всего в силлогистике для четырех типов высказываний **A, I, E** и **O** можно получить 256 различных модусов – правил вывода. В общем случае различают следующие рассуждения:

- *индуктивные* – от частного к общему;
- *достоверные – дедуктивные*, от общего к частному;
- *правдоподобные* – от частного к частному.

Индуктивные рассуждения от частного к общему отражают путь познания окружающего мира. Общие утверждения возникают при попытке обобщения частных, отражающих совокупность единичных фактов, полученных из опыта. Истинность общего утверждения будет очевидной, если частных утверждений, подтверждающих результат, будет достаточно много и не будет опровергающих утверждений.

Полной индукцией называют рассуждения, в которых общее заключение о принадлежности некоторого свойства или признака предметам данного класса делается на основании принадлежности данного свойства или признака всем предметам данного класса. Полная индукция дает истинное знание при условии, что граница рассматриваемого класса объектов точно известна.

Неполная индукция – это перенос знаний, известных о части объектов данного класса, на все объекты данного класса. Она основывается на свойстве повторяемости признаков у сходных предметов. Однако здесь могут возникать ошибочные индуктивные заключения из-за применения второстепенных признаков в качестве существенных, т.е. в индуктивных рассуждениях из истинных посылок могут получаться ложные заключения.

Правдоподобные индуктивные рассуждения достигаются не только на основе выявления повторяемости признаков у объектов некоторого класса, но и их взаимосвязи и причинной зависимости между данными признаками и свойствами рассматриваемых объектов.

В индуктивных выводах используются различные методы установления причинно-следственных отношений. Формулируются они в виде принципов, основными из которых являются ***принципы:*** *единственного различия, единственного сходства, единственного остатка, аналогии* и другие.

Например, формулировка *принципа единственного различия*: ”Если после введения какого-либо фактора появляется, или после удаления его исчезает, известное явление, причем мы не вводим и не удаляем никакого другого обстоятельства, которое могло бы изменить в данном случае влияние, и не производим никакого изменения среди первоначальных условий явления, то указанный фактор и составляет причину явления”. Этот принцип можно описать следующим образом: Пусть в серии из n опытов A, B, C вызывают Д; в другой серии из n опытов B, C не вызывают Д. Тогда на основании наблюдений можно сделать следующий вывод: "Вероятно, А является причиной Д".

Применяются также ***нечеткие выводы***, когда истинность посылок принимается с некоторой степенью уверенности и заключение также выводимо из таких посылок с определенной степенью достоверности (вероятности). Нечеткие выводы имеют под

собой математическое обоснование в виде теории нечетких множеств, основанной Л.Заде.

Основные идеи, лежащие в основе ***дедуктивных рассуждений***, восходят к работам Аристотеля и состоят в следующем:

1) исходные посылки рассуждения являются истинными;
2) правильно применяемые приемы перехода от посылок к вытекающим из них утверждениям и из посылок и ранее полученных утверждений к новым вытекающим из них утверждениям должны сохранять истинность получаемых утверждений – истинные посылки порождают истинные следствия.

Наиболее важные практические результаты в СИИ получены при использовании дедуктивных рассуждений, на базе которых построено большинство логических систем представления знаний.

20.3. Представление знаний

Представление знаний – это соглашение о том, как описывать реальный мир. В естественных и технических науках принят следующий традиционный способ представления знания. На естественном языке вводятся основные понятия и отношения между ними. При этом используются ранее определенные понятия и отношения, смысл которых уже известен. Далее устанавливается соответствие между характеристиками (чаще всего количественными) понятий знания и подходящей математической модели.

Основная цель представления знания – строить математические модели реального мира и его частей, для которых соответствие между системой понятий проблемного знания может быть установлено на основе совпадения имен переменных модели и имен понятий без предварительных пояснений и установления дополнительных неформальных соответствий. Представление знаний обычно выполняется в рамках той или иной системы представления знаний.

Системой представления знаний (СПЗ) называют средства, позволяющие описывать знания о предметной области с помощью языка представления знаний, организовывать хранение знаний в системе (накопление, анализ, обобщение и структурирование знаний), вводить новые знания и объединять их с имеющимися, выводить новые знания из имеющихся, находить требуемые знания, устранять устаревшие знания, проверять непротиворечивость накопленных знаний, осуществлять интерфейс между пользователем и знаниями.

Центральное место в СПЗ занимает ***язык представления знаний*** (ЯПЗ). В свою очередь, выразительные возможности ЯПЗ определяются лежащей в основе ЯПЗ моделью представления знаний (иногда эти понятия отождествляют).

Модель представления знаний является формализмом, призванным отобразить статические и динамические свойства предметной области (ПО), т. е. отобразить объекты и отношения ПО, связи между ними, иерархию понятий ПО и изменение отношений между объектами.

Модель представления знаний может быть универсальной (применимой для большинства ПО) или специализированной (разработанной для конкретной ПО). В СИИ используются следующие *основные универсальные модели представления знаний*:

- семантические сети;
- фреймы;
- системы продукций;

- логические модели и другие.

Во всех разработанных системах с базами знаний, кроме этих моделей, взятых за основу, использовались специальные дополнительные средства. Тем не менее, классификация моделей представления знаний остается неизменной.

Семантические сети

Семантические сети (СС) являются исторически первым классом моделей представления знаний. Здесь структура знаний предметной области формализуется в виде ориентированного графа с размеченными вершинами и дугами. Вершины обозначают сущности и понятия ПО, а дуги – отношения между ними. Под сущностью понимают объект произвольной природы. Вершины и дуги могут снабжаться метками, представляющими собой мнемонические имена. Основными связями для СС, с помощью которых формируются понятия, являются:

- класс, к которому принадлежит данное понятие;
- свойства, выделяющие понятие из всех прочих понятий этого класса;
- примеры данного понятия.

На самой СС принадлежность элемента к некоторому классу или части к целому передается с помощью связок “IS A” и “PART OF” соответственно. Свойства описываются связками “IS” и “HAS” (“является” и “имеет”). На рис. 20.1 приведен пример описания понятия с помощью СС.

Рис. 20.1. Пример фрагмента семантической сети

С помощью СС можно описывать события и действия. Для этих целей используются специальные типы отношений, называемые падежами: агент – действующее лицо, вызывающее действие; объект – предмет, подвергающийся действию; адресат – лицо, пользующееся результатом действия или испытывающее этот результат. Возможны и другие падежи типа: время, место, инструмент, цель, качество, количество и т. д. Введение падежей позволяет от поверхностной структуры предложения перейти к его смысловому содержанию.

В СС понятийная структура и система зависимостей представлены однородно. Поэтому представление в них, например, математических соотношений графическими средствами неэффективно. СС не дают ясного представления о структуре ПО, они представляют собой пассивные структуры, для обработки которых необходима разработка аппарата формального вывода и планирования.

В чистом виде СС на практике почти не используются. При построении СИИ с использованием СС обычно либо накладывают ограничения на типы объектов и отношений (примером таких сетей являются функциональные СС), либо расширяют СС специальными средствами для более эффективной организации вычислений в СС (К-сети, пирамидальные сети и др.).

Фреймы

Метод представления знаний с помощью фреймов предложен М. Минским. Фрейм – это структура, предназначенная для представления стереотипной ситуации. Каждый фрейм описывает один концептуальный объект, а конкретные свойства этого объекта и факты, относящиеся к нему, описываются в слотах – структурных элементах данного фрейма. Все фреймы взаимосвязаны и образуют единую фреймовую систему, в которой объединены и процедурные знания.

Концептуальному представлению свойственна иерархичность, целостный образ знаний строится в виде единой фреймовой системы, имеющей иерархическую структуру. В слот можно подставить разные данные: числа или математические соотношения, тексты, программы, правила вывода или ссылки на другие слоты данного или других фреймов.

Фрейм определяется как структура следующего вида:

(ИМЯ ФРЕЙМА;
ИМЯ СЛОТА1 (ЗНАЧЕНИЕ СЛОТА1)
ИМЯ СЛОТА2 (ЗНАЧЕНИЕ СЛОТА2)
.....
ИМЯ СЛОТАN (ЗНАЧЕНИЕ СЛОТАN))

Определим, например, фрейм для объекта “Служащий”:

(Служащий
ФИО (Петров И.П.)
Должность (инженер)
Категория (2)
.....)).

Если значения слотов не определены, то фрейм называют *фреймом-прототипом*, в противном случае – *конкретным фреймом* или *экземпляром фрейма.*

В теории фреймов ничего не говорится о методах реализации фрейма. Вслед за появлением теории фреймов появилось целое семейство систем программирования, поддерживающих концепцию фрейм-подхода: **KRL, GUS, FRL, OWL** и другие.

Для большинства фреймовых языков свойственно иерархическое описание объектов предметной области с использованием типовых фреймов. При этом широко используется механизм наследования свойств одного объекта (представленных в виде значений слотов связанного с ним фрейма) другими объектами. Используются такие виды наследования, как класс-подкласс, класс- экземпляр класса. Это позволяет согласовать однотипную информацию различных объектов, а также в дальнейшем обеспечить соответствующее их поведение.

Фреймовые системы относят к процедуральной форме представления знаний. Объясняется это тем, что управление выводом во фреймовых системах реализуется путем подключения так называемых присоединенных процедур, разрабатываемых пользователем.

Процедуры связываются со слотами и обычно именуются демонами и слугами. ***Демон*** – это процедура, которая активизируется автоматически, когда в ее слот подставляется значение или проводится сравнение значений. ***Слуга*** – это процедура, которая активизируется по запросу – при возникновении определенного события.

С использованием присоединенных процедур можно запрограммировать любую процедуру вывода на фреймовой сети. Механизм управления выводом организуется

следующим образом. Сначала запускается одна из присоединенных процедур некоторого фрейма, называемого образцом. Образец – это, по сути, фрейм–прототип, т. е. у него заполнены не все слоты, а только те, которые описывают связи данного фрейма с другими. Затем в силу необходимости, посредством пересылки сообщений, последовательно запускаются присоединенные процедуры других фреймов и, таким образом, осуществляется вывод.

Язык представления знаний, основанных на фреймовой модели, эффективен для структурного описания сложных понятий и решения задач, в которых в соответствии с ситуацией желательно применять различные способы вывода. В то же время на таком языке затрудняется управление завершенностью и постоянством целостного образа. В частности, по этой причине существует опасность нарушения присоединенной процедуры, проблема зацикливания процесса вывода.

Отметим, что фреймовую модель без механизма присоединенных процедур, а, следовательно, и механизма пересылки сообщений, часто используют как базу данных системы продукций.

Продукционные системы

Продукционные системы – это системы представления знаний, основанные на правилах типа

“УСЛОВИЕ-ДЕЙСТВИЕ”.

Записываются эти правила обычно в виде

ЕСЛИ $A_1,A_2,...,A_n$ ТО B.

Такая запись означает, что "если выполняются все условия от A_1 до A_n (являются истинными), тогда следует выполнить действие **B**". Часть правила после **ЕСЛИ** называется ***посылкой***, а часть правила после **ТО – *выводом,*** или ***действием***, или ***заключением***.

Условия $A_1,A_2,...,A_n$ обычно называют ***фактами***. С помощью фактов описывается текущее состояние предметной области. Факты могут быть истинными, ложными либо, в общем случае, правдоподобными, когда истинность факта допускается с некоторой степенью уверенности.

Действие **B** трактуется как добавление нового факта в описание текущего состояния предметной области.

В упрощенном варианте описание ПО с помощью правил (продукций) базируется на следующих основных предположениях об устройстве предметной области. ПО может быть описана в виде множества фактов и множества правил.

Факты – это истинные высказывания (в естественном языке – это повествовательные предложения) об объектах или явлениях предметной области.

Правила описывают причинно-следственные связи между фактами (в общем случае и между правилами тоже) – как истинность одних фактов влияет на истинность других. Такое представление предметной области является во многих случаях достаточным, а вот соответствует ли оно действительному положению вещей, зависит от точки зрения наблюдателя.

Описание ПО нетрудно ввести в ЭВМ – для этого достаточно снабдить его соответствующими средствами для хранения множества фактов (например, в виде базы фактов), для хранения правил (например, в базе правил), и построить

интерпретатор базы правил, который по описанию текущего состояния ПО в виде предъявленных ему фактов осуществляет поиск выводимых из фактов заключений.
На этой идее и построены системы продукций. Типичная структура системы, основанной на правилах, приведена на рис. 20.2.

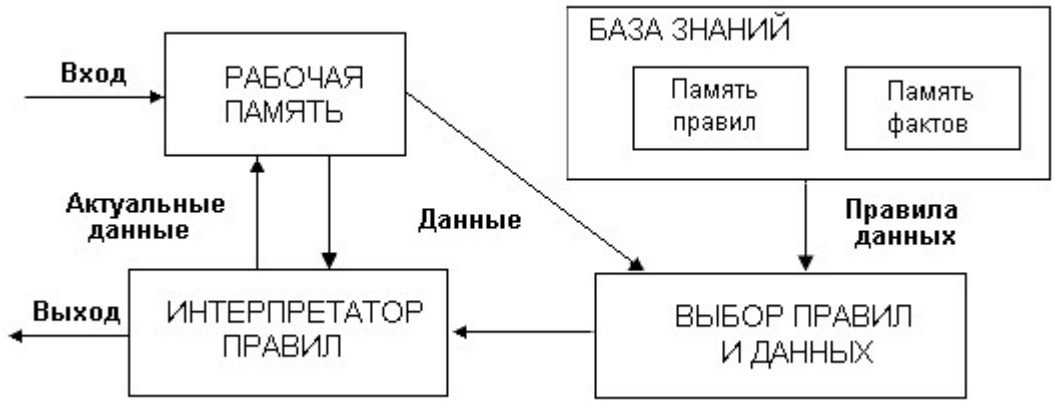


Рис. 20.2. Структура продукционной системы

В продукционных системах используются два основных способа реализации механизма вывода:
1) прямой вывод, или вывод от данных;
2) обратный вывод, или вывод от цели.
В первом случае идут от известных данных (фактов) и на каждом шаге вывода к этим фактам применяют все возможные правила, которые порождают новые факты и так до тех пор, пока не будет порожден факт-цель.
Для применения правила используется процесс сопоставления известных фактов с правилами и, если факты согласуются с посылками в правиле, то правило применяется.
Во втором случае вывод идет в обратном направлении – от поставленной цели. Если цель согласуется с заключением правила, то посылку правила принимают за подцель или гипотезу, и этот процесс повторяется до тех пор, пока не будет получено совпадение подцели с известными фактами.
Рабочая память представляет собой информационную структуру для хранения текущего состояния предметной области. Обмен информацией в продукционной системе осуществляется через рабочую память. К примеру, из одного правила нельзя переслать какие–либо данные непосредственно в другое правило, минуя рабочую память. Состояние рабочей памяти целиком определяет подмножество применимых на каждом шаге вывода правил.
Например, возможная формулировка правил продукций в экспертной системе диагностики автомобиля имеет вид.

Если (горит_лампа_датчика_давления_масла
и уровень_масла_норма
и обороты_двигателя_норма
и масляный фильтр_не_засорен)
То (проверить масляный насос)

Приведенное правило позволяет принять решение по ремонту системы смазки автомобиля.

Достоинством применения правил продукций является их модульность. Это позволяет легко добавлять и удалять знания в базе знаний. Можно изменять любую из продукций, не затрагивая содержимого других продукций.
Недостатки продукционных систем проявляются при большом числе правил и связаны с возникновением непредсказуемых побочных эффектов при изменении старых и добавлении новых правил. Кроме того, отмечают также низкую эффективность обработки систем продукций и отсутствие гибкости в логическом выводе.

Логические системы

В основе логических систем представления знаний лежит понятие формальной логической системы. Оно является также одним из основополагающих понятий формализации. Основные идеи ***формализации*** заключаются в следующем. Вводится множество базовых элементов (алфавит) теории. Определяются правила построения правильных объектов (предложений) из базовых элементов. Часть объектов объявляется изначально заданными и правильными по определению – аксиомами. Задаются правила построения новых объектов из других правильных объектов системы (правила вывода).
Данная схема лежит в основе построения многих ***дедуктивных*** СИИ. В соответствии с ней база знаний описывается в виде предложений и аксиом теории, а механизм вывода реализует правила построения новых предложений из имеющихся в базе знаний. На вход СИИ поступает описание задачи на языке этой теории в виде запроса (предложения, теоремы), которое явно не представлено в БЗ, но если оно верно с позиций заложенных в БЗ знаний и не противоречит им, то может быть построено из объектов БЗ путем применения правил вывода. Процесс работы механизма вывода называют ***доказательством запроса*** (теоремы). Если запомнить шаги процесса вывода в виде трассы и представить ее пользователю, то она будет объяснением выработанного СИИ решения задачи.
Формальные языки, на которых записываются предложения (формулы) с использованием рассмотренных понятий, получили названия ***логических языков***. С практической и теоретической точек зрения наиболее важными и изученными являются *язык логики высказываний* и *язык логики предикатов*. В языке логики высказываний элементарные предложения рассматриваются как неделимые сущности, в языке логики предикатов, наоборот, делается расчленение предложения на субъект и предикат.
В процессе математизации рассуждений различают два вида слов: ***термы*** – аналоги имен существительных и ***формулы*** – аналоги повествовательных предложений.
Для записи предложений используются стандартные формы высказываний, что даёт с одной стороны, стандартизовать рассуждения, т.е. рассматривать только определённые структуры посылок и заключений, а с другой ввести в термы *переменные* – именные формы, которые обращаются в имена после подстановки вместо переменных конкретных значений.
Формулы с переменными, обращающиеся в высказывания при подстановке вместо переменных значений, называют ***высказывательными формами*** или переменными высказываниями. Одна форма порождает множество истинных или ложных высказываний.
Однако не все предложения, содержащие переменные, являются высказывательными формами. Различают связанные и свободные переменные. Так, сложные предложения с переменными, содержащие логические связки “СУЩЕСТВУЕТ” или

”ВСЕ”, обозначают высказывания, а переменные, к которым они относятся, являются связанными.

Расчленение предложения на субъект и предикат в математической логике математизируется путем соотнесения предложения, выражающего свойства предмета, с функцией одной переменной **P(x)**. При этом сама функция **P** – логическая функция одной переменной, т.е. ***одноместный предикат***, а аргумент **x** – ***субъект***. Если же предложение описывает отношение между несколькими (**n**) субъектами, то с ним можно связать **n**-местную логическую функцию **P(x1,x2,...,xn)** – **n**-местный предикат.

Логические связки “И”, ”ИЛИ”, ”НЕ” и т. д., с помощью которых строятся сложные предложения (формулы) соотносятся с операциями логики следующим образом:

неверно что – $\neg$ (знак отрицания);
и – $\wedge$ (знак конъюнкции);
или – $\vee$ (знак дизъюнкции);
если ... то – $\rightarrow$ (знак импликации);
тогда, когда – $\Leftrightarrow$ (знак эквивалентности).

Логические связки “ДЛЯ ВСЯКОГО”, ”СУЩЕСТВУЕТ” относятся к переменным в предложении и обозначают:

для всякого – $\forall$ – знак квантора общности;
существует – $\exists$ – знак квантора существования.

В различных логических системах используются разнообразные правила вывода. Приведем два наиболее распространенных из них.

Первое – "***правило подстановки***" имеет следующую формулировку. В формулу, которая уже выведена, можно вместо некоторого высказывания подставить любое другое при соблюдении условия: подстановка должна быть сделана во всех местах вхождения заменяемого высказывания в данную формулу.

Второе – "***правило заключения***" (латинское название ***Modus ponens*** – положительный модус) состоит в следующем: Если α и $\alpha \rightarrow \beta$ являются истинными высказываниями посылками, тогда и высказывание заключение β также истина. Записывается правило в виде дроби:

$$\frac{\alpha,\ \alpha \rightarrow \beta}{\beta}$$

Пример. Пусть имеются следующие истинные высказывания:

1. Если самолет проверен и заправлен, то он готов к вылету.
2. Если самолет готов к вылету и дано разрешение на взлет, то он либо взлетел, либо находится на взлётной полосе.
3. Если самолет взлетел, то он выполняет рейс.
4. Самолет ЯК-42 проверен и заправлен.
5. Самолет ТУ-134 проверен.
6. Самолет ИЛ-62 заправлен.
7. Самолету ЯК-42 дано разрешение на вылет.
8. Самолет ЯК-42 не находится на взлетной полосе.

Требуется найти, какой из самолетов в момент времени T выполняет рейс.

Проведем анализ данных высказываний. Высказывания 1, 2, 3 являются сложными и построены с использованием логических связок $\rightarrow$ (импликация), $\wedge$ (И). Во всех элементарных высказываниях, из которых построены предложения 1, 2, 3, субъектом является понятие “самолёт”; предикатами выступают сказуемые, описывающие свойства всех объектов, принадлежащих классу “самолет”. Высказывания 4-8

являются фактами, истинными на момент времени Т. Они являются элементарными высказываниями, описывающими свойства конкретных объектов предметной области.

Для формального описания задачи введем следующие одноместные предикаты:

ПРОВЕРЕН(X) – самолет X проверен;

ЗАПРАВЛЕН(X) – самолет X заправлен;

ГОТОВ(X) – самолет X готов к вылету;

ДАНО_РАЗР(X) – самолету X дано разрешение на вылет;

ВЗЛЕТЕЛ(X) – самолет X взлетел;

НАХ_ВЗП(X) – самолет X находится на взлетной полосе;

НЕ_НАХ_ВЗП(X) – самолет X не находится на взлетной полосе;

ВЫП_РЕЙС(X) – самолет X выполняет рейс.

Тогда исходное описание на языке логики предикатов будет иметь вид:

1. $\forall X$(ПРОВЕРЕН(X)$\wedge$ ЗАПРАВЛЕН(X) $\rightarrow$ ГОТОВ(X))
2. $\forall X$(ГОТОВ(X) $\wedge$ДАНО_РАЗР(X)$\wedge$НЕ_НАХ_ВЗП(X) $\rightarrow$ ВЗЛЕТЕЛ(X))
3. $\forall X$(ГОТОВ(X) $\wedge$ДАНО_РАЗР(X)$\wedge\neg$ ВЗЛЕТЕЛ(X) $\rightarrow$ НАХ_ВЗП(X))
4. $\forall X$(ВЗЛЕТЕЛ(X) $\rightarrow$ ВЫП_РЕЙС(X))
5. ПРОВЕРЕН(ЯК-42)
6. ЗАПРАВЛЕН(ЯК-42)
7. ПРОВЕРЕН(ТУ-134)
8. ЗАПРАВЛЕН(ИЛ-62)
9. ДАНО_РАЗР(ЯК-42)
10. НЕ_ НАХ_ВЗП(ЯК-42)

Предложения 1-4, хотя и содержат переменную, являются высказываниями – переменная X связана квантором общности $\forall$. В дальнейшем квантор писать не будем, так как он присутствует во всех предложениях.

Чтобы найти, какой из самолетов в момент времени Т выполняет рейс, подготовим запрос вид:

$$M \Rightarrow \text{ВЫП\_РЕЙС}(Z),$$

где M – множество предложений 1-10.

Вывод запроса можно представить следующей последовательностью шагов.

1 шаг.

Применив к предложению 1 подстановку X=ЯК-42, получим заключение ПРОВЕРЕН(ЯК-42)$\wedge$ ЗАПРАВЛЕН(ЯК-42) $\rightarrow$ ГОТОВ(ЯК-42).

2 шаг.

Первая посылка: объединив предложения 5 и 6, получим ПРОВЕРЕН(ЯК-42)$\wedge$ЗАПРАВЛЕН(ЯК-42) .

Вторая посылка: заключение шага 1: ПРОВЕРЕН(ЯК-42) $\wedge$ ЗАПРАВЛЕН(ЯК-42) $\rightarrow$ ГОТОВ(ЯК-42).

Применив правило Modus Ponens

$$\frac{\alpha,\ \alpha\rightarrow\beta}{\beta}$$

для α= ПРОВЕРЕН(ЯК-42) $\wedge$ ЗАПРАВЛЕН(ЯК-42) и β= ГОТОВ(ЯК-42), получим следующее заключение: ГОТОВ(ЯК-42).

3 шаг.

Первая посылка: объединив заключение шага 2, предложения 9 и 10, получим: ГОТОВ(ЯК-42) ∧ ДАНО_РАЗР(ЯК-42) ∧ НЕ_ НАХ_ВЗП(ЯК-42).
Вторая посылка: применив к правилу 2 подстановку X=ЯК-42, получим
ГОТОВ(ЯК-42) ∧ДАНО_РАЗР(ЯК-42)∧НЕ_НАХ_ВЗП(ЯК-42) → ВЗЛЕТЕЛ(ЯК-42)
Применив правило Modus Ponens, получим заключение ВЗЛЕТЕЛ(ЯК-42).
4 шаг.
Первая посылка: заключение шага 3 – ВЗЛЕТЕЛ(ЯК-42).
Вторая посылка: применив к правилу 4 подстановку X=ЯК-42, получим
ВЗЛЕТЕЛ(ЯК-42) → ВЫП_РЕЙС(ЯК-42).
Применив правило Modus Ponens, получим заключение ВЫП_РЕЙС(ЯК-42).
Таким образом, в момент времени T рейс выполняет самолет ЯК-42. Остальные подстановки, например X=ИЛ-62, приводят к тупиковым ситуациям. Логический вывод выполнялся нами в прямом направлении, при этом в процессе вывода трижды использовалось правило заключения.

20.4. Направления работ и инструментарий ИИ

В настоящее время исследования в области ИИ имеют следующую прикладную ориентацию:

- общение на естественном языке и моделирование диалога;
- экспертные системы (ЭС);
- автоматическое доказательство теорем;
- робототехника;
- интеллектуальные пакеты прикладных программ;
- распознавание образов;
- решение комбинаторных задач.

Наибольшие практические результаты достигнуты в создании ЭС, которые получили уже широкое распространение и используются при решении практических задач.
Экспертная система представляет собой программный комплекс, содержащий знания специалистов из определенной предметной области, обеспечивающий консультациями менее квалифицированных пользователей для принятия экспертных решений. Основное отличие ЭС от обычных программ, также способных поддерживать экспертные решения, заключается в отделении декларативных знаний от манипулирующего знаниями процедурного компонента.
Структура экспертной системы зависит от ее назначения и решаемых задач. В состав современных экспертных систем (рис. 20.3) входят следующие *основные компоненты*: база знаний, решатель, редактор базы знаний, подсистема объяснений и интерфейс пользователя.

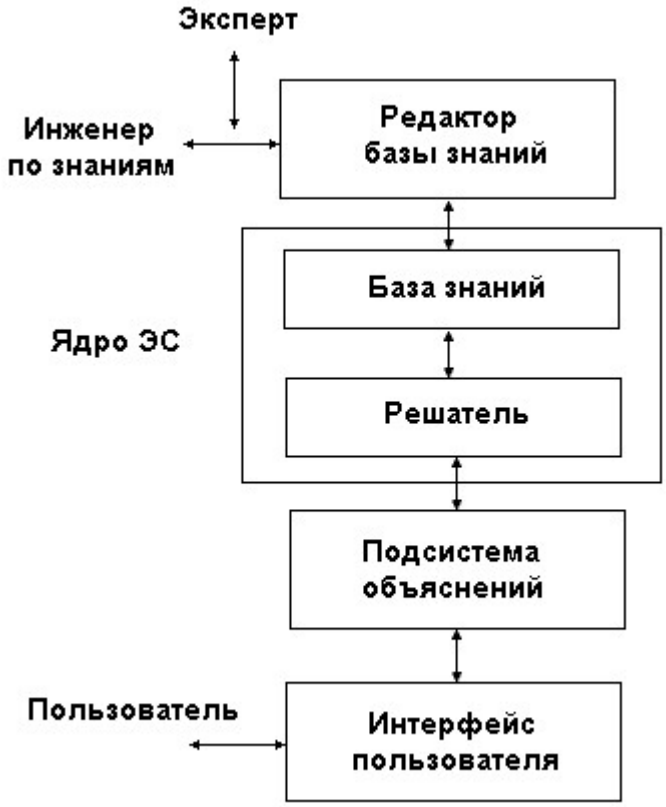


Рис. 20.3. Структура экспертной системы

Определение и взаимодействие компонентов ЭС может быть описано следующим образом.

База знаний представляет собой совокупность знаний о предметной области, организованных в соответствии с принятой моделью представления знаний.

Решатель, или подсистема логического вывода представляет собой программу, обеспечивающую автоматический вывод решения формулируемых пользователем или экспертом задач на основе знаний, хранящихся в базе.

База знаний и решатель вместе составляют основную часть – ***ядро ЭС***. В ряде источников к ядру ЭС относят только базу знаний.

Инженер по знаниям – специалист по искусственному интеллекту, помогающий эксперту вводить знания в базу знаний.

Эксперт – специалист в предметной области, способный принимать экспертные решения и формулирующий знания о предметной области для ввода их в базу знаний.

Редактор базы знаний – это программа, предназначенная для ввода в базу знаний новых знаний о предметной области для представления их в базе знаний.

Пользователь ЭС является специалистом в данной предметной области, квалификация которого уступает квалификации эксперта.

Интерфейс пользователя есть комплекс программ, обеспечивающих удобный диалог с ЭС при вводе запросов на решение экспертных задач и получении результатов.

Подсистема объяснений представляет собой программу, которая позволяет пользователю выполнить трассировку логического вывода и получить мотивировку умозаключений на каждом этапе цепочки вывода.

Конкретная экспертная система создается в результате совместной работы инженера по знаниям и эксперта. Взаимодействие пользователя с ЭС осуществляется через интерфейс пользователя на близком к естественному или профессиональному языку предметной области непроцедурном языке. При этом производится трансляция предложений на язык представления знаний (ЯПЗ) экспертной системы. Описание запроса на ЯПЗ поступает в решатель, в котором на основе знаний из базы выводится решение поставленного запроса в соответствии с некоторой стратегией выбора правил. С помощью подсистемы объяснений производится отображение промежуточных и окончательных выводов, объяснение применяемой мотивировки.

Интеллектуальный пакет прикладных программ (ИППП) можно определить как интегрированную систему, позволяющую пользователю решать задачи без программирования – путем описания задачи и исходных данных. Программирование осуществляется автоматически программой планировщиком из набора готовых программных модулей, относящихся к конкретной предметной области. В числе примеров ИППП можно назвать систему ПРИЗ, в которой пользователь формирует свою задачу на непроцедурном языке УТОПИСТ. Еще одним примером ИППП является система СПОРА, в которой формирование задачи пользователь выполняет на непроцедурном языке ДЕКАРТ.

К числу ИППП относятся решатели вычислительных задач. Ниже мы рассмотрим решатель вычислительных задач TK Solver, с помощью которого можно описывать и решать задачи вычислительного характера без программирования.

Инструментальные средства СИИ

Используемые для разработки СИИ, в том числе ЭС, инструментальные средства можно разделить на следующие типы:

- системы программирования на языках высокого уровня;
- системы программирования на языках представления знаний;
- оболочки систем искусственного интеллекта – скелетные системы;
- средства автоматизированного создания ЭС.

Системы программирования на языках высокого уровня (ЯВУ) таких как C++, Паскаль, Фортран, Бэйсик, Forth, Refal, SmallTalk, Лисп и др. в наименьшей степени ориентированы на решение задач искусственного интеллекта. Они не содержат средств, предназначенных для представления и обработки знаний. Тем не менее, достаточно большая, но со временем снижающаяся, доля СИИ разрабатывается с помощью традиционных ЯВУ. В приведенном перечне можно выделить языки Лисп и SmallTalk, как наиболее удобные и широко используемые для создания СИИ. В частности, широкое использование языка Лисп объясняется наличием развитых средств работы со списками и поддержкой механизма рекурсии, важных для характерной в СИИ обработки символьной информации. С помощью языка Лисп разработан ряд распространенных ЭС, таких как MYCIN, DENDRAL, PROSPECTOR.

Системы программирования на языках представления знаний имеют специальные средства, предназначенные для создания СИИ. Они содержат собственные средства представления знаний (в соответствии с определенной моделью) и поддержки логического вывода. К числу языков представления знаний можно отнести FRL, KRL, OPS5, LogLisp, Пролог и др. Разработка СИИ с помощью систем программирования на ЯПЗ основана на технологии обычного программирования. От разработчика требуются соответствующие программистские

навыки и квалификация. Наибольшее распространение из числа названных языков получили язык логического программирования Пролог и OPS5.

Средства автоматизированного создания ЭС представляют собой гибкие программные системы, допускающие использование нескольких моделей представления знаний, способов логического вывода и видов интерфейса, и содержащие вспомогательные средства создания ЭС. В качестве примеров рассматриваемого класса средств можно назвать следующие системы: EXSYS (предназначена для создания прикладных ЭС классификационного типа), 1st-Class, Personal Consultant Plus, ПИЭС (программный инструментарий экспертных систем), GURU (интегрированная среда разработки ЭС), Xi Plus, OPS5+. Построение ЭС с помощью рассматриваемых средств заключается в формализации исходных знаний, записи их на входном языке представления знаний и описании правил логического вывода решений. Далее экспертная система наполняется знаниями.

К рассматриваемому классу систем можно отнести также ***специальный программный инструментарий***. К примеру, сюда относятся библиотеки и надстройки над языком Лисп: KEE (Knowledge Engeneering Environment – среда инженерии знаний), FRL (Frame Represantation Language – язык представления фреймов), KRL (Knowledge Represantation Language – язык представления знаний) и др. Они повышают возможности и гибкость в работе с заготовками экспертных систем.

Оболочки, или "пустые" экспертные системы представляют собой готовые экспертные системы без базы знаний. Примерами оболочек ЭС, получивших широкое применение, являются зарубежная оболочка EMYCIN (Empty MYCIN – пустой MYCIN) и отечественная оболочка Эксперт-микро, ориентированная на создание ЭС решения задач диагностики. Технология создания и использования оболочки ЭС заключается в том, что из готовой экспертной системы удаляются знания из базы знаний, затем база заполняется знаниями, ориентированными на другие приложения. Достоинством оболочек является простота применения – специалисту нужно только заполнить оболочку знаниями, не занимаясь созданием программ. Недостатком применения оболочек является возможное несоответствие конкретной оболочки разрабатываемой с ее помощью прикладной ЭС.

21. ПРОГРАММИРОВАНИЕ В СИСТЕМЕ ТУРБО-ПРОЛОГ

21.1. Пролог и логическое программирование

Логическое программирование (ЛП) – это направление в программировании, основанное на идеях и методах математической логики. Термин “логическое программирование” в литературе по информатике трактуется в широком и узком смысле.

В широком толковании в ЛП включают круг понятий, методов, языков и систем, в основе которого лежит идея описания задачи совокупностью утверждений на некотором логическом языке и получение решения задачи путем построения логического вывода в некоторой формальной дедуктивной системе. Классы формул, используемые для описания задач, методы вывода и модели вычислений, используемые в таких системах очень разнообразны. Поэтому различные их

комбинации образуют специфические стили программирования: концептуальное, функциональное (аппликативное) и другие.

Наибольшие практические результаты достигнуты в системах программирования, где в качестве логических программ используются специальные классы логических формул – хорновские дизъюнкты, а в качестве способа их использования применяются специальные методы логического вывода – варианты метода резолюций. Программирование с использованием таких систем называют ***хорновским*** и ***резолюционным***, но чаще всего – ***логическим программированием***, перенося узкую трактовку термина на более широкое понятие. Самыми известными системами такого рода являются реализации языка Пролог (Программирование в терминах логики – Programming in logic). Рассмотрим основные идеи и понятия ЛП в узком смысле.

Неформально логическая программа описывает множество объектов, множество функций и отношений на этих объектах. Строится логическая программа как набор утверждений об объектах, функциях и отношениях. Такое описание является статическим и никакого вычислительного процесса не задает, т.е. можно считать, что это описание определяет *базу данных*, в которой хранятся объекты и заданные на них функции и отношения, например, в виде графиков.

Конкретному применению логической программы соответствует понятие *запроса* (цели) – например, каково значение функции, заданной логической программой, при данном значении аргумента. Вычисление ответа на запрос соответствует доказательству существования такого объекта. Правила, по которым проводятся вычисления, образуют процедурную – операционную семантику логической программы. Успехи языка Пролог обусловлены тем, что с одной стороны, с помощью используемых в нем логических формул – хорновских дизъюнктов можно описать многие практические задачи, а с другой – найдена простая интерпретация этих формул и построена достаточно эффективная реализация системы логического программирования.

Правило в Прологе имеет вид:

$$\mathbf{A}_0 \leftarrow \mathbf{A}_1,...,\mathbf{A}_m \ (\mathbf{m \geq 0}),$$

где $\mathbf{A}_0,\mathbf{A}_1,...,\mathbf{A}_m$ - атомы. Атом $\mathbf{A}_0$ называется *заголовком*, а $\mathbf{A}_1,...,\mathbf{A}_m$ - *телом* правила. Тело может быть пустым (при m=0) – такие правила называют *фактами.* Атом имеет вид

$$\omega(\mathbf{t}_1,\mathbf{t}_2,...,\mathbf{t}_n),$$

где ω – n-арный предикатный символ или имя отношения; $\mathbf{t}_1,\mathbf{t}_2,...,\mathbf{t}_n$ - термы.

Терм – это либо имя переменной, либо константа, либо составной терм вида $\mathbf{f(t}_1,\mathbf{t}_2,...,\mathbf{t}_n\mathbf{)}$, **f – n**-арный функциональный символ. Функции, задаваемые логической программой, представляются в виде отношений – **n**-местная функция $\mathbf{y=f(x}_1,\mathbf{x}_2,...,\mathbf{x}_n\mathbf{)}$ представляется в виде **(n+1)**-местного отношения вида $\mathbf{F(x}_1,\mathbf{x}_2,...,\mathbf{x}_n,\mathbf{y)}$.

Запрос (цель) имеет вид:

$$\leftarrow \mathbf{C}_1,\mathbf{C}_2,...,\mathbf{C}_r,$$

где $\mathbf{r \geq 0}$ и $\mathbf{C}_1,\mathbf{C}_2,...,\mathbf{C}_r$ – атомы.

Каждое правило допускает логическую и процедурную интерпретации (семантики). Логическая интерпретация правила $\mathbf{A_0 \leftarrow A_1,...,A_m}$ – “истинность $\mathbf{A_0}$ следует из истинности $\mathbf{A_1}$ и истинности $\mathbf{A_2}$, и..., истинности $\mathbf{A_m}$“ или “ $\mathbf{A_0}$ истинно, если

истинны $\mathbf{A_1}$ и $\mathbf{A_2}$, и..., и $\mathbf{A_m}$".Таким образом, правила рассматриваются как формулы языка логики предикатов вида:

$$\forall x_1, \forall x_2, ..., \forall x_n \, (\mathbf{A_1} \wedge \mathbf{A_2} \wedge ... \wedge \mathbf{A_m} \rightarrow \mathbf{A_0}).$$

Здесь $\forall$ – квантор общности, $\wedge$ – логическая связка И, $\rightarrow$ – логическая связка ЕСЛИ-ТО.

В Прологе, в силу традиции, данные формулы записываются в обратную сторону и используются другие обозначения для логических связок: $\wedge$ обозначается "," (запятой) или словом **and**; $\vee$ обозначается ";" (точкой с запятой) или словом **or** ; $\rightarrow$ обозначается в теории $\leftarrow$, а в языке программирования используется конструкция ":-" (двоеточие и минус) или слово **if**. Все переменные в предложениях связаны квантором общности, поэтому обычно знак квантора $\forall$ опускается.

Формула с использованием связок $\rightarrow$ и $\wedge$ может быть преобразована в эквивалентную, но имеющую связки $\neg$ (НЕ) и $\vee$ (ИЛИ), формулу вида:

$$\mathbf{A_0} \vee \neg \mathbf{A_1} \vee \neg \mathbf{A_2} \vee ... \vee \neg \mathbf{A_m} \quad (\mathbf{m \geq 0})$$

Такие формулы и называются ***хорновскими дизъюнктами***, а стиль программирования – ***хорновским***.

Процедурная интерпретация правил Пролога вида:

$\mathbf{A_0}$**:-** $\mathbf{A_1}$**,** $\mathbf{A_2}$**, ...,** $\mathbf{A_m}$ или $\mathbf{A_0}$ **if** $\mathbf{A_1}$ **and** $\mathbf{A_2}$ **and ...** **and** $\mathbf{A_m}$ следующая: "для достижения цели $\mathbf{A_0}$ необходимо последовательно достичь целей $\mathbf{A_1}$, $\mathbf{A_2}$, ..., $\mathbf{A_m}$ ".

Соответственно, для фактов (когда **m=0**) имеет место логическое прочтение: "$\mathbf{A_0}$ истинно ", процедурная интерпретация факта: " цель $\mathbf{A_0}$ достигнута".

Логическая семантика запроса **G**: $\leftarrow \mathbf{C_1}, \mathbf{C_2}, ..., \mathbf{C_r}$ (с переменными $\mathbf{x_1}, \mathbf{x_2}, ..., \mathbf{x_l}$) к логической программе $\mathfrak{R}$ понимается как требование вычислить все значения переменных $\mathbf{x_1}, \mathbf{x_2}, ..., \mathbf{x_l}$, при которых утверждение $\mathbf{C_1} \wedge \mathbf{C_2} \wedge ... \wedge \mathbf{C_r}$ логически следует из утверждений программы $\mathfrak{R}$ или, записывая это, как принято в логике, необходимо показать, что

$$\mathfrak{R} \Rightarrow \mathbf{G}$$

т. е. запрос **G** логически следует (выводим) из логической программы $\mathfrak{R}$. При этом оказывается, что доказательство выводимости **G** из $\mathfrak{R}$ аналогично тому, чтобы показать

$$\mathfrak{R}, \neg \mathbf{G} \Rightarrow$$

т. е. присоединение отрицания запроса **G** к программе $\mathfrak{R}$ приводит к противоречию. Однако последнее выполнить в классе формул, с помощью которых записывается Пролог-программа, существенно проще и имеется эффективный метод доказательства – ***метод резолюций***, одна из версий которого и применяется в Прологе.

Метод резолюций, получив на вход логическую программу с присоединенным к ней запросом в виде множества хорновских дизъюнктов, пытается построить вывод пустого дизъюнкта, обозначаемого символом . Если это ему удается, тогда цель допускается, в противном случае отвергается. Реализуется метод резолюций с помощью двух правил: правила *резолюции* и правила *склейки*, которые во время работы вызывают процедуру *унификации* – сопоставления.

Унификация – процесс, на вход которого поступает два терма и для них находится унификатор. **Унификатором** двух термов называется подстановка, которая делает термы одинаковыми. Если унификатор существует, то термы называются

унифицируемыми и для них отыскивается наиболее общий унификатор, если нет – процедура унификации сообщает об отказе.

Пусть имеются два терма t_1=ОТЕЦ(X,Y) и t_2=ОТЕЦ(“ПЕТР”,”ПАВЕЛ”). Унификатором этих двух термов будет подстановка Θ=(X=“ПЕТР”, Y=”ПАВЕЛ”), т.е. если в терм t_1 вместо переменных X и Y подставить значения из соответствующих мест терма t_2, термы станут одинаковыми, а значением их будет пример ОТЕЦ(“ПЕТР”,”ПАВЕЛ”).

Пусть теперь t_1=ОТЕЦ(X,”ИВАН”) и t_2=ОТЕЦ(“ПЕТР”,”ПАВЕЛ”). В этом случае унификация невозможна, так как с помощью подстановки термы нельзя сделать одинаковыми – на втором месте в обоих термах стоят разные константные выражения. Если бы термы были, например, вида t_1=ОТЕЦ(X,”ИВАН”) и t_2=ОТЕЦ(“ПЕТР”,Z), тогда Θ=(X=“ПЕТР”, Z=”ИВАН”), и пример для этих термов ОТЕЦ(“ПЕТР”,”ИВАН”).

Правило резолюции позволяет из дизъюнктов

$\mathbf{D_1 = \neg A_1(t) \vee A_2 \vee ... \vee A_m}$;

$\mathbf{D_2 = A_1(s) \vee B_1 \vee ... \vee B_n}$;

получить новый дизъюнкт $\mathbf{D=\Theta(A_2 \vee ... \vee A_m \vee B_1 \vee ... \vee B_n)}$.

Здесь **Θ –** наиболее общий унификатор термов **t** и **s,** обеспечивает их равенство и означает, что все подстановки унификатора выполнены для всех атомов, входящих в дизъюнкты $\mathbf{D_1}$ и $\mathbf{D_2}$. Дизъюнкты $\mathbf{D_1}$ и $\mathbf{D_2}$ называют родителями дизъюнкта **D**. В дизъюнкте **D** отсутствует пара:$\mathbf{\neg A_1(\Theta t) \vee A_1(\Theta s)}$, при этом **(Θt)= (Θs)** и пара является тавтологией (тождественно-истинной) и может быть удалена из дальнейших вычислений, что и выполняет правило резолюции.

Правило склейки позволяет из дизъюнкта $\mathbf{A(t) \vee A(s) \vee B_1 \vee ... \vee B_n}$ получить дизъюнкт $\mathbf{\Theta(A(t) \vee B_1 \vee ... \vee B_n)}$, т. е. осуществляется “склеивание” одинаковых атомов, полученных после унифицирующей подстановки **Θt=Θs**.

Работу метода резолюции опишем на примере, приведенном ранее. Запишем множество утверждений и фактов в виде логической программы, используя Прологообразный синтаксис:

1 ГОТОВ(X1) **if** ПРОВЕРЕН(X1) **and** ЗАПРАВЛЕН(X1).

2.ВЗЛЕТЕЛ(X2) **if** ГОТОВ(X2) **and** ДАНО_РАЗР(X2) **and** (НЕ_НАХ_ВЗП(X2).

3. НАХ_ВЗП(X3) **if** ГОТОВ(X3) **and**ДАНО_РАЗР(X3) **and not**(ВЗЛЕТЕЛ(X3).

4. ВЫП_РЕЙС(X4) **if** (ВЗЛЕТЕЛ(X4).

5. ПРОВЕРЕН(ЯК-42)

6.ЗАПРАВЛЕН(ЯК-42)

7. ПРОВЕРЕН(ТУ-134)

8. ЗАПРАВЛЕН(ИЛ-62)

9. ДАНО_РАЗР(ЯК-42)

10.НЕ_ НАХ_ВЗП(ЯК-42)

Предложения 1-4 являются утверждениями Пролога, предложения 5-10 – факты. Переменные в каждом утверждении являются локальными – их область действия одно утверждение.

Представим утверждения в форме *дизъюнктов*:

1. ГОТОВ(X1) $\vee\neg$ ПРОВЕРЕН(X1) $\vee\neg$ ЗАПРАВЛЕН(X1).
2. ВЗЛЕТЕЛ(X2) $\vee\neg$ ГОТОВ(X2)$\vee\neg$ ДАНО_РАЗР(X2) $\vee \neg$ НЕ_НАХ_ВЗП(X2).

3. НАХ_ВЗП(X3) ∨ ¬ ГОТОВ(X3) ∨ ¬ДАНО_РАЗР(X3) ∨ ВЗЛЕТЕЛ(X3)
4. ВЫП_РЕЙС(X4)∨¬ ВЗЛЕТЕЛ(X4)
5. ПРОВЕРЕН(ЯК-42)
6. ЗАПРАВЛЕН(ЯК-42)
7. ПРОВЕРЕН(ТУ-134)
8. ЗАПРАВЛЕН(ИЛ-62)
9. ДАНО_РАЗР(ЯК-42)
10.НЕ_ НАХ_ВЗП(ЯК-42)

Запрос:?¬ ВЫП_РЕЙС(Z). Доказательство запроса приведено ниже.

Запрос	Правило	Подстановка
?¬ВЫП_РЕЙС(Z)	4	Z=X4
?¬ВЗЛЕТЕЛ(X4)	2	X4=X2
?¬ГОТОВ(X2)∨¬ДАНО_РАЗР(X2)∨ ¬НЕ_НАХ_ВЗП(X2)	1	X2=X1
?¬ПРОВЕРЕН(X1)∨¬ ЗАПРАВЛЕН(X1)∨ ¬ДАНО_РАЗР(X1) ∨¬ НЕ_НАХ_ВЗП(X1)	5	X1=ЯК-42
?¬ЗАПРАВЛЕН(ЯК-42)∨¬ДАНО_РАЗР(ЯК-42) ∨ ¬ НЕ_НАХ_ВЗП(ЯК-42)	6	
?¬ДАНО_РАЗР(ЯК-42)∨ ¬НЕ_НАХ_ВЗП(ЯК-42)	9	
?¬ НЕ_НАХ_ВЗП(ЯК-42)	10	
□		

В результате доказательства мы вывели логически пустой дизъюнкт □, следовательно, цель допускается. Композиция подстановок Z=X4=X2=X1=ЯК-42 на наш запрос дает ответ Z=ЯК-42. Любая другая подстановка, например, X1=ТУ-134, приводит в тупик.

21.2. Работа в системе программирования

Система программирования Турбо-Пролог включает в свой состав:

- интегрированную среду для разработки, отладки и выполнения логических программ;
- язык логического программирования Турбо-Пролог.

Рассмотрим коротко интерфейс интегрированной среды и основы программирования в этой среде на языке Турбо-Пролог.

Интерфейс системы Турбо-Пролог

Рассматриваемая версия системы логического программирования Турбо-Пролог 2.0 работает под управлением DOS и имеет стандартизованный программный интерфейс. Главное меню интегрированной среды системы включает в свой состав следующие меню и команды: Files (Файл), Edit (Правка), Run (Выполнение), Compile (Компиляция), Options (Параметры) и Setup (Настройка). Дадим краткую характеристику команд в составе указанных меню.

Меню ***Files (Файл)*** содержит следующие команды управления файлами программ на Прологе: Load (Загрузка файла), Pick (Загрузка файла), New file (Создание файла), Save (Сохранение файла), Write to (Сохранение файла с новым именем), Directory

(Вывод текущего каталога), Change dir (Смена каталога), OS Shell (Временный выход в среду DOS) и Exit (Выход).
Команда ***Edit (Правка)*** служит для переключения в режим редактирования исходного текста программы на Прологе.
Команда ***Run (Выполнение)*** запускает процесс компиляции и выполнения текущей исходной программы на языке Турбо-Пролог.
Меню ***Compile (Компиляция)*** содержит следующие команды управления компиляцией: Memory (Компиляция программы с размещением в оперативной памяти), OBJ file (Компиляция с созданием объектного файла), EXE file (auto link) (Построение файла EXE), Project (Создание файла проекта), Link only (Редактирование связей файла объектной программы OBJ), Link options (Параметры редактирования связей), EDIT PRJ file (Правка файла проекта), Compiler directives (Директивы компилятора).
Меню ***Setup (Настройка)*** содержит следующие команды настройки системы: Colors (Цвета), Window Size (Размеры окон), Directories (Управление каталогами), Miscellaneous (Параметры внешних устройств), Load SYS file (Загрузить файл SYS), Save SYS file (Сохранить файл SYS).
В нижней части диалогового окна системы программирования Турбо-Пролог содержится строка подсказки о назначении функциональных клавиш. Справочная помощь системы вызывается нажатием клавиши <F1>.

Основы программирования

Алфавит языка Турбо-Пролог включает следующие символы:

- прописные и строчные буквы латинского алфавита (**A-Z, a-z**);
- цифры (**0-9**);
- специальные символы: **! @ # $ & ( ) | ^ * – + / < > ;, ? . \ _ “ ‘ ~**.

В Прологе имена используются для обозначения символических констант, доменов, предикатов и переменных. В общем случае имя должно начинаться с буквы или знака подчеркивания, за которым идет любая комбинация букв, цифр или знаков подчеркивания. При этом правила именования различных типов объектов в Пролог-программе имеют свои особенности:

- имена символических констант и имена предикатов должны начинаться со *строчной* буквы;
- имена предметных переменных (аргументы предикатов и функций) должны начинаться с *прописной* буквы или знака подчеркивания.

В качестве имени переменной в предикатах может использоваться знак подчеркивания, такая переменная называется *анонимной,* применяется это в случаях, когда значение переменной безразлично для Пролог-программы.
К *ключевым* (служебным) словам в языке Турбо-Пролог относятся следующие слова:

and	**domains**	**goal**	**include**
clauses	**elsedef**	**if**	**or**
constants	**enddef**	**ifdef**	**predicates**
database	**global**	**ifndef**	

Программа на Турбо-Прологе имеет следующую структуру:

```
/*-----------------------------------------------------*/
/*              Комментарии                            */
```

/*--*/
constants
/*определение констант */
domains
/*определение типов данных программы*/
database
/*определение предикатов динамической базы данных*/
predicates
/*определение предикатов*/
clauses
/*определение правил и фактов*/
goal
/*определение целей*/
/*--*/
/* Комментарии */
/*--*/

В разделе **constants** объявляются используемые в программе константы.

В разделе **domains** объявляются нестандартные типы данных для переменных, используемых в качестве аргументов предикатов. В Прологе типы данных называют *доменами*. Связывание типа домена с конкретным аргументом (местом) предиката осуществляется в секции **predicates**. Домен описывает множество значений, которые может принимать переменная предиката в ходе выполнения программы.

Домены подразделяются на простые и структурированные, стандартные и нестандартные. К стандартным относятся:

symbol – символьная константа (длина не более 250 символов), имеет две формы записи: последовательность букв, цифр и знаков подчеркивания, начинающаяся со *строчной* буквы; последовательность символов, заключенная в *двойные кавычки.* Примеры: apple, sort1,”personal”,”Курсант Петров С.В.”

string – строка символов: любая последовательность символов, заключенная в двойные кавычки.

char – отдельный символ, заключенный между двумя апострофами.

integer – целое число в диапазоне от-32768 до +32767.

real – действительное число, допускается обычная и экспоненциальная формы записи. Значение экспоненты должно быть в диапазоне от E-307 до E+308. Примеры: -34.567, 0.654, 9.76e+3.

file – файловая переменная, значение ее определяется по правилам именования файлов и устройств в MS-DOS. При выполнении операций с файлом, ее необходимо связать с конкретным файлом или устройством**.**

Объявление новых доменов с использованием стандартных имеет вид:

<ИМЯ>=<ИМЯ СТАНДАРТНОГО ДОМЕНА>

Примеры объявлений:

domains
a=integer
fas=symbol
ret,das=real

Такие объявления новых доменов улучшают читабельность программы и обеспечивают контроль типов значений переменных – смешивать в ходе выполнения программы переменные разных типов (доменов) нельзя.

Кроме стандартных типов доменов, в Турбо-Прологе допускается использовать *структуры* доменов, состоящие из нескольких простых или сложных объектов. ***Объявление структуры*** имеет следующий вид:

<СТРУКТУРА>=<ИМЯ СТРУКТУРЫ>(<Д1>,<Д2>,...,<ДN>)

Здесь **<ИМЯ СТРУКТУРЫ>** называют *функтором*, а домены **<Д1>,<Д2>...,<ДN> –** это либо простые домены, либо имена ранее объявленных доменов, либо, в свою очередь, структуры. Структуры позволяют сортировать объекты по категориям. Ссылки на доменную структуру осуществляются по имени функтора.

В одном объявлении можно описать несколько альтернативных вариантов структуры, разделяя варианты точкой с запятой или служебным словом **or**.

Примеры объявления структур:

```
domains
  d1,d2,d3=symbol
  fr=fruits(d1,d2,d3) ; pot(d1)
```

Раздел описания предикатов **predicates** содержит перечень предикатов пользователя, используемых в программе. Описание предиката содержит имя предиката и список доменов его аргументов:

<ИМЯ ПРЕДИКАТА>(<Д1>,<Д2>,...,<ДN>)

Здесь **<Д1>,<Д2>,...,<ДN>**- имена стандартных доменов или имена доменов, объявленных в разделе **domains**.

Один и тот же предикат может иметь различное число аргументов, такие предикаты объявляются для каждого варианта отдельно. В программе допускается использовать не более 300 предикатов, число аргументов у предиката не должно превышать 50.

Примеры объявления предикатов:

```
predicates
        add(integer,integer,integer)
        lk(fr)
        lk(d1,d2)
```

В разделе **database** описываются предикаты динамической базы данных. Перечисленные здесь предикаты после подстановки в них вместо переменных констант (т.е. превращения их в *факты*) могут быть помещены и, если потребуется, удалены во время выполнения программы в динамическую базу данных. Делается это с помощью стандартных (встроенных в систему) предикатов: **assert, asserta, assertz, consult, retract, retractall.** В программе можно использовать несколько разделов **database**, при этом каждому из них можно назначить уникальное имя. Если имя разделу **database** не назначено, компилятор по умолчанию назначает имя **dbasedom**.

Раздел **database** имеет следующий формат:

```
database [-<имя базы данных>]
        dbpred1 (...)
        dbpred2 (...)
```

В разделе **clauses** описываются утверждения, каждое из которых является правилом или фактом. В конце каждого утверждения ставится точка.

Факт состоит из имени предиката и заключенного в скобки списка аргументов – констант.

Правило состоит из заголовка – предиката, объявленного в разделе **predicates,** за которым следует двоеточие с дефисом (**:-**), а затем список вызовов предикатов (пользовательских и/или стандартных), разделенных запятыми или точками с

запятой. Вместо двоеточия с дефисом можно использовать ключевое слово **if,** вместо запятой – ключевое слово **and**, вместо точки с запятой – ключевое слово **or**.
Правила и факты, имеющие в качестве заголовка один и тот же предикат, должны быть сгруппированы в рамках одного блока, т. е. следовать в программе друг за другом.
Переменные в предикатах во время выполнения Пролог-программы могут находиться в двух состояниях: *конкретизированном* или *свободном* (неконкретизированном). Переменная является свободной, если ей не присвоено значение, в противном случае переменная является конкретизированной.
В теле правила, кроме объявленных в программе предикатов, могут использоваться *стандартные* предикаты и *операции сравнения*.
Стандартные предикаты выполняют разнообразные функции по вводу-выводу, работе с файлами, выполнению функций DOS, обработке строк, поддержке графического режима, обеспечению интерфейса с другими системами программирования и т.д. Описание основных стандартных предикатов Турбо-Пролога рассматривается ниже.
В правилах можно использовать следующие основные ***операции сравнения***: **<** (меньше)**, >** (больше), **<=** (меньше или равно)**, >=** (больше или рано)**, =** (равно)**, <>** или **><** (не равно). Сравнивать между собой можно выражения и переменные.
Операция = (равно) устанавливает соответствие между выражениями правой и левой частей предиката X=Y – предикат этот записан в привычной инфиксной форме. В процессе *согласования* переменных используются следующие *соглашения*:
1) если X – свободная переменная, а Y– конкретизированная, то при записи X=Y, X станет конкретизированной и получит значение, равное Y;
2) целые числа и строки всегда равны самим себе;
3) две структуры равны, если они имеют одинаковые функторы, одинаковое число параметров и все соответствующие параметры равны между собой.
4) если имеется запись вида X=Y и обе переменные свободны, то они становятся *сцепленными* и при конкретизации одной из них, вторая автоматически будет означена тем же значением.

Раздел **goal** содержит *запрос* к программе, называемый *внутренним*. Для внутреннего запроса Пролог осуществляет поиск только *первого подходящего решения.* При этом система не сообщает о результатах найденного решения (успешное или нет). Полученные при сопоставлении значения переменных, входящих в предикаты запроса также не отображаются на экране. Эти действия должен запрограммировать программист с использованием стандартных предикатов для вывода данных.
Раздел описания целей (**goal)** в компилируемой программе может отсутствовать, тогда в диалоговом окне после запуска программы на выполнение можно ввести ***внешний запрос***. При использовании внешнего запроса Турбо-Пролог отыскивает *все варианты решений* и в этом же окне выводятся значения переменных предикатов запроса и сообщение об успешном или не успешном решении.

21.3. Управление вычислениями

Основными средствами управления процессом вычислений в Прологе являются стандартные предикаты **fail** (*неуспех*) и **!** (*отсечение*).
Назначение этих предикатов и методы их использования рассмотрим на примере следующей программы на Турбо-Прологе:
domains

```
  st=student(fam,pr,oc)
  fam,pr=symbol
  num,oc=integer
  g=gr(num,st)
predicates
  kurs_22(g)
clauses
  kurs_22(gr(261,student("ПЕТРОВ П.Р.","Программирование",5))).
  kurs_22(gr(261,student("ИВАНОВ Б.О.","Операционные системы",5))).
  kurs_22(gr(261,student("СИДОРОВ Т.К.","Системы управления",4))).
  kurs_22(gr(262,student("ЖИГАРЕВ С.И.","Программирование",3))).
  kurs_22(gr(262,student("ДЕМИН С.Л.","Системы управления",5))).
  kurs_22(gr(261,student("ПЕТРОВ П.Р.","Иностранный язык",4))).
  kurs_22(gr(263,student("СИДОРОВ  Е.Р.","Операционные системы",5)))
```

Приведенная программа в разделе **clauses** содержит утверждения-факты, в данном случае информацию о результатах пересдачи экзаменов студентами соответствующих групп (номер группы – первый компонент структуры **gr**) по соответствующим дисциплинам (фамилия, дисциплина, оценка – компоненты структуры **student**, которая входит в состав структуры **gr**). Если теперь после компиляции программы в качестве *внешней* цели ввести запрос:

```
kurs_22(X)
```

то в диалоговом окне будет выведена информация:

```
X=gr(261,student("ПЕТРОВ П.Р.","Программирование",5))
X= gr (261,student("ИВАНОВ Б.О.","Операционные системы",5))
X= gr (261,student("СИДОРОВ Т.К.","Системы управления",4))
X= gr (262,student("ЖИГАРЕВ С.И.","Программирование",3))
X= gr (262,student("ДЕМИН С.Л.","Системы управления",5))
X= gr (261,student("ПЕТРОВ П.Р.","Иностранный язык",4))
X= gr (263,student("СИДОРОВ  Е.Р.","Операционные системы",5))
7 Solutions
```

т. е. будут найдены все варианты решений, так как в запросе не задано ограничений на значения переменной **X**.

Наложим ограничение, например, на номер учебной группы. Для внешнего запроса:

```
kurs_22(gr(261,X))
```

результатом будет:

```
X=student("ПЕТРОВ П.Р.","Программирование",5)
X=student("ИВАНОВ Б.О.","Операционные системы",5)
X=student("СИДОРОВ Т.К.","Системы управления",4)
X=student("ПЕТРОВ П.Р.","Иностранный язык",4)
4 Solutions
```

Продемонстрируем еще один внешний запрос:

```
kurs_22(gr(Z,student("ПЕТРОВ П.Р.", X,Y)))
```

система ответит:

```
Z=261, X=Программирование, Y=5
Z=261, X=Иностранный язык, Y=4
2 Solutions
```

Если теперь любой из этих запросов сделать *внутренним*, т. е. записать в разделе **goal**, откомпилировать и выполнить программу, то, кроме сообщения Press the

SPACE bar, в диалоговом окне не будет результатов, так как мы не задавали предикатов вывода значений результата внутреннего запроса.
Изменим внутренний запрос:

goal

kurs_22(X),write(X)

Здесь для вывода значений переменной **X** после сопоставления используется системный предикат **write(...).** Результаты такого запроса будут помещены в диалоговое окно в виде:

gr(261,student("ПЕТРОВ П.Р.","Программирование",5))

т. е. будет найдено *первое подходящее решение*. Если теперь, не меняя запроса, изменить порядок предложений в программе, например, поменять местами первое со вторым, то результат будет другим:

gr (261,student("ИВАНОВ Б.О.","Операционные системы",5))

Как видим, Пролог чувствителен к порядку предложений в программе: правила просматриваются сверху вниз. Чтобы заставить Пролог осуществить поиск всех вариантов решения задачи при использовании *внутренней цели* в программе (в данном варианте это вывод на экран всех фактов), необходимо организовать циклический перебор альтернативных решений. В Прологе это можно сделать различными способами.
Один из них – использовать свойство Пролога выполнять поиск альтернативного решения при неудачной попытке применения текущего. В этом случае Пролог выполняет *откат* до ближайшей альтернативы, восстанавливает первоначальные условия поиска и процесс начинается заново с найденной новой альтернативы. Для организации таких вычислений в языке есть специальный предикат **fail**, который всегда завершается неуспехом и вызывает откат до ближайшей альтернативы.
Для цели перебора всех альтернативных фактов в примере и вывода их на экран в состав программы введем нульместный предикат *show* и правило вида*: show:- kurs_22(X),write(X),fail*. Для этого в раздел **predicates** добавим строку *show*, в разделе **clauses** поместим строку *show:- kurs_22(X),write(X),fail***.**
Зададим также *внутреннюю* цель вида:

goal

show.

В этом случае на экран будут выведены все 7 альтернативных решений – после нахождения первого переменная **X** будет означена константой первого факта, полученное значение переменной **X** сопоставляется аргументу предиката *write(X),* который выведет его на экран. После этого выполняется предикат ***fail,*** который завершается неуспехом, и Пролог выполнит откат до ближайшего недетерминированного предиката (т. е. имеющего несколько вариантов правил), а это в нашем случае предикат *kurs_22(X),* будет выбран второй факт, выполнено сопоставление и т. д. до тех пор, пока не будут перебраны все альтернативы.
Предписать Прологу выполнять *откат* можно не только с помощью предиката ***fail*** **–** любой неуспешно завершенный предикат вызывает откат до ближайшей альтернативы (стоящего от него левее в теле правила). Продемонстрируем это на нашем примере. Пусть необходимо получить информацию и вывести ее на экран о студентах, пересдавших экзамен по дисциплине "Программирование".
Изменим предикат *show:*

```
show:- kurs_22(gr(X,student(Y,Z,K))),
        Z="Программирование ",
        write("Группа-",X," Студент-",Y," Оценка-",K),
```

nl,
fail.

Тогда для *внутреннего* запроса:

goal
show

на экран будет выведено:

Группа-261 Студент-ПЕТРОВ П.Р. Оценка-5
Группа-262 Студент-ЖИГАРЕВ С.И. Оценка-3

Таким образом, сравнение Z=”Программирование ”, выполняет двоякую роль: с одной стороны это *фильтр*, который не допускает выполнения предиката ***write(...)*** при неуспешном сравнении, а с другой – вызывает из-за неуспешного сравнения *откат* до ближайшей альтернативы. При этом если бы не было предиката **fail**, было бы получено только первое решение – ***fail*** обеспечивает перебор всех альтернатив. Предикат ***nl*** является стандартным, он выполняет переход на следующую строку при выводе информации на экран.

В Турбо-Прологе имеются средства, с помощью которых можно заблокировать поиск с возвратом. Для этих целей служит специальный предикат, который называется **“отсечение”** (cut) и обозначается восклицательным знаком. Для демонстрации его использования в нашем примере добавим предикат отсечения в правило *show:*

show:- kurs_22(gr(X,student(Y,Z,K))),
Z=”Программирование ”,!,
write(“Группа-”,X,” Студент-”,Y,” Оценка-”,K),
nl,
fail.

Тогда для внутреннего запроса:

goal
show

на экране будет выведена информация:

Группа-261 Студент-ПЕТРОВ П.Р. Оценка-5

В данном варианте после успешного сравнения *Z=”Программирование ”* предикат **!** прекращает условия поиска после формирования условия “неуспех” предикатом ***fail***. Таким образом, использование в правиле предиката отсечения означает, что в дальнейшем не будет производиться перебор других аргументов предикатов, использованных в этом правиле *до знака отсечения* после формирования признака “неуспеха ” любым из предикатов, стоящим в правиле правее предиката **!**. Но это не означает, что не будут перебираться альтернативные варианты по неуспеху для других предикатов, стоящих за **!**.

Преобразуем предикат *show* к виду:

show:- kurs_22(gr(X,student(Y,Z,K))),
Z=”Программирование ”,!,K=3,
write(“Группа-”,X,” Студент-”,Y,” Оценка-”,K),
nl,
fail.

В этом случае не будет получено результатов, так как после поиска первого решения для *Z=”Программирование ”* переменная *K=5* и сравнение *5#3* сформирует неуспех, который не приведет к возврату для поиска следующей альтернативы для предиката *kurs_22(gr(X,student(Y,Z,K))),* так как условия возврата уже сброшены отсечением.

Введем в правило еще один предикат *kurs_22(gr(A,student(B,C,D))),* который зависит от других переменных – это принципиально. Предикат *show* будет следующим:

```
show:- kurs_22(gr(X,student(Y,Z,K))),
       Z=”Программирование ”,!,
       kurs_22(gr(A,student(B,C,D))),D=3,B=Z,
       write(“Группа-”,A,” Студент-”,B,” Оценка-”,D),
       nl, fail.
```

После внутреннего запроса *show* на экран будет выведена информация в виде:

Группа-262 Студент-ЖИГАРЕВ С.И. Оценка-3

Отметим, что часть правила после отсечения **!** осуществляет полный перебор всех фактов программы, начиная с первого, с помощью фильтра *D=3,B=Z,* а при его выполнении – с помощью ***fail***.

Другими словами, отсечение ограничивает распространение аргументов для подбора новых значений возврата (бэктрекинга) из-за неудачи. Отсечение применяется также для ускорения вычислений путем отбрасывания избыточных ветвей вычислений.

Замечания:

- Ранее было показано отличие выполнения *внутреннего* (описанного в разделе goal) и *внешнего* (вводимого через приглашение системы в диалоговом окне) запросов. Теперь мы можем сказать, что перебор всех альтернативных решений для *внешнего* запроса обеспечивается автоматическим присоединением к нему системой Турбо-Пролог стандартного предиката ***fail***.
- Наличие в программе раздела **goal** позволяет после компиляции логической программы получить загрузочный модуль программы (файл типа .EXE) и использовать его независимо от среды программирования Турбо-Пролог.

21.4. Рекурсивные вычисления

Использование рекурсии обеспечивает организацию циклических вычислений в Прологе. Рекурсия применяется обычно в ситуациях, когда число возможных решений заранее не известно, либо когда обрабатываются структуры данных с произвольным числом элементов.

Рекурсивное описание правила содержит в своем теле ссылку на заголовок этого же правила. При этом возможны следующие варианты рекурсивных правил:

1) правая рекурсия

   pr1()*:-pr11(),pr12(),…,****pr1()****.*

2) левая рекурсия

   pr1()*:-****pr1()****,pr21(),pr22(),…*

3) обобщенная рекурсия

   pr1()*:-pr11(),pr12(),…,****pr1()****,pr21(),pr22(),….*

Для того чтобы во время выполнения рекурсивного правила не происходило зацикливания, необходимо предусмотреть условия завершения рекурсии. Их можно реализовать двумя способами:

1) заданием в программе альтернативного правила или факта ***pr1()****,* не содержащего рекурсии (выход произойдет при успешном выполнении этого альтернативного правила);

2) формированием условия выхода одним из предикатов *pr11(),pr12()*,…, – выход происходит, если в процессе выполнения правила хотя бы один из предикатов завершается неуспехом.

Предикаты *pr21(),pr22(),…*не влияют на выполнение рекурсии – они выполняются только после выхода из нее и получают значения переменных из стека, в который они помещаются во время выполнения рекурсии. Производимые при этом вычисления называют **хвостовыми вычислениями**.

Основные идеи реализации рекурсивных определений в Прологе рассмотрим на примере вычисления факториала. Программа на Прологе имеет вид:

```
domains
  number,product=integer
predicates
  fact(number,product)
clauses
  fact(1,1):-!.
  fact(N,R):-Next_N=N-1,
             fact(Next_N,P),
             R=N*P.
goal
  fact(3,Res),write("Факториал 3= ",Res),nl.
```

Действия при выполнении программы по шагам представлены в табл. 21.1.

Таблица 21.1.

Действия при выполнении программы

Вызов предиката	**Подстановки**	**Вычисления**	**Хвостовые вычисления**	**Результаты вычислений**
fact(3,Res)	N=3, Res=R,	Next_N=N-1=2	R=N*P	Res=R=3*2=6
fact(2,P)	N'=2, P=R'	Next_N'=N'-1=2	R'=N'P'	P=R'=2*1=3
fact(1,P')	1=1, P'=1			

На первом шаге выполняется целевой запрос **fact(3,Res)** и выбирается первый вариант решения: правило **fact(1,1):-!**, сопоставление для него заканчивается неуспехом, так как **3#1**, происходит откат и выбирается второе правило. Для него сопоставление с заголовком правила заканчивается успешно, при этом выполняется подстановка **N=3, Res=R**, тело правила помещается в стек, и начинают отрабатываться предикаты тела. Первый из них – сравнение **Next_N=N-1,** оно завершается успешно, переменная **Next_N** становится равной **2.**

Затем инициируется вызов **fact(2,P),** снова выбирается первый вариант правила, сопоставление **1#2** приводит к неуспеху, откат и выбор второго варианта правила. Здесь сопоставление с заголовком правила заканчивается успешно, при этом выполняется подстановка **N'=2, P'=R',** новая копия правила помещается в стек и начинается его выполнение. Первое из них – сравнение **Next_N'=N'-1**, завершается успешно, копия переменной **Next_N'** становится равной **2-1=1**.

Затем выполняется вызов **fact(1,P').** Снова производится сопоставление с первым вариантом, и вот здесь оно становится успешным, подстановка имеет вид **1=1, P'=1,** тело первого правила пусто, поэтому никаких действий не инициирует и начинается выполнение "хвостов", оставшихся в стеке от копий второго правила. В результате этих вычислений переменная **Res** получит значение равное **6,** которое и будет выведено на экран следующим предикатом запроса **write("Факториал 3= ",Res).**

Отметим, что в первом утверждении применено отсечение **!**. Сделано это для того,

чтобы в случае использования внешнего запроса вида **fact(1,Res),** т. е. запроса на вычисление факториала 1, после успешного сопоставления запроса с первым утверждением (1=1,Res=1) отсечь второй вариант правила для нахождения альтернативного решения. Если этого не сделать, то второе правило приведет к зацикливанию и ошибке. Для обеспечения корректной работы программы для запроса **fact(0,Res)** в раздел **clauses** нужно добавить еще одно утверждение, а именно **fact(0,1):-!.**

21.5. Списки

Список – это упорядоченный набор объектов – элементов списка, следующих друг за другом. Элементы списка в языке Турбо-Пролог должны принадлежать одному и тому же доменному типу. *Объектами списка* могут быть: целые числа, действительные числа, символы, символьные строки, структуры.

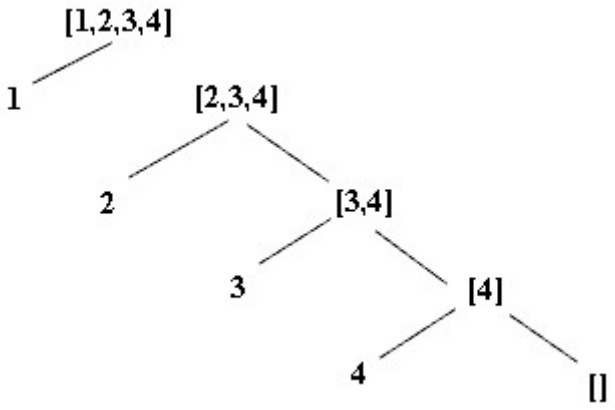
Совокупность элементов списка заключается в квадратные скобки, а элементы друг от друга отделяются запятыми. Примерами списков являются:

[10,24,1,8,385,0,8]
[0.4,67.43,986.01,914.5]
[“ПЕТРОВ П.Р.”,“ИВАНОВ Б.О.”,“СИДОРОВ Т.К.”]

Список может содержать произвольное число элементов (ограничением является объем доступной оперативной памяти). Список, не содержащий элементов, называется пустым списком и обозначается []. Обработка списков в Турбо-Прологе базируется на разбиении списка на голову – первый элемент в списке и хвост – остальная часть списка. Например, как показано в следующей таблице.

Список	**Голова**	**Хвост**
[1,2,3,4]	1	[2,3,4]
[‘a’,’b’,’c’]	’a’	[‘b’,’c’]
[“Курсант”]	”Курсант”	[]
[]	не определено	не определено

Наглядным представлением процесса разбиения списка на голову и хвост является графическое представление в виде *бинарного* (*двоичного*) *дерева*. Для списка [1,2,3,4] таким деревом будет следующее:



Для использования списков в Пролог-программе необходимо выполнить следующее.

1) В разделе ***domains*** описать домен списка. Домен списка задается в формате:

<ИМЯ ДОМЕНА>=<ТИП ЭЛЕМЕНТА>*

Например, домен списка, элементами которого являются целые числа, описывается следующим образом:

```
domains
        list_num=integer*
```

или

```
        list_num=elem*
        elem=integer
```

2) Описать работающий со списком предикат в разделе **predicates**, например:

```
predicates
    num(list_num)
```

3) Ввести или создать в программе собственно список, т.е. задать список в разделах **goal** или **clauses**.

Пример задания списков в Пролог-программе:

```
domains
    list1=symbol*
    list2=num*
    num=integer
predicates
    name(list1)
    score(list2)
clauses
    name(["ПЕТРОВ П.Р.","ИВАНОВ Б.О.","СИДОРОВ Т.К."]).
    score([1,3,5,7,9]).
```

Ниже приведены возможные примеры *внешних* запросов к программе и получаемые при этом результаты:

Запрос	**Результат**
name(X)	*X= ["Петров П.Р.","Иванов Б.О.","Сидоров Т.К."].*
name([_,Y,_])	*Y="Иванов Б.О."*
score([A,B,_,_,C])	*A=1,B=3,C=9*

При использовании в Пролог-программе списков с произвольным числом элементов, используется метод разделения списка на *голову* и *хвост.* Этот метод обеспечивает рекурсивную обработку списка. Операция разделения списка на голову и хвост в языке Турбо-Пролог обозначается вертикальной чертой (|) и записывается в виде:

[Head | Tail]

Здесь: **Head** – переменная для обозначения головы списка; **Tail** – переменная для обозначения хвоста списка.

Пример 1. Рассмотрим программу распечатки содержимого списка, элементы которого могут быть целыми числами или символьными строками:

```
domains
  list1=integer*
  list2=symbol*
predicates
  print_list(list1)
  print_list(list2)
clauses
  print_list([]).
  print_list([Head|Tail]):- write(Head),
  nl,
```

```
  print_list(Tail).
goal
  print_list([1,2,3,4]).
```

В данной программе предикат списка *print_list()* описан для двух типов доменов, поэтому с его помощью могут обрабатываться списки, элементами которых являются как целые числа, так и символьные строки. В программе реализован рекурсивный метод обработки.

Первое правило программы *print_list([])* описывает тот факт, что пустой список не нужно печатать, оно же является условием выхода из рекурсии, используемым во втором правиле – печать списка завершается тогда, когда список пуст. Опишем работу программы.

Первоначально аргумент целевого внутреннего запроса **print_list([1,2,3,4])** сопоставляется с первым вариантом правила; сопоставление терпит неуспех, так как [1,2,3,4]#[]. Затем выбирается второй вариант правила, здесь сопоставление завершается успехом и выполняется операция | – разделения целевого списка на голову и хвост. Результатом ее является означение Head=1 и Tail=[2,3,4]. Далее начинают последовательно выполняться предикаты тела правила.

Первым в теле стоит стандартный предикат **write(Head)**, который выводит на экран значение переменной **Head**, в данном случае 1, затем выполняется предикат **nl –** перевод строки и формируется рекурсивный вызов **print_list([2,3,4])**. Снова обращение к первому варианту правила и снова неуспех – [2,3,4]#[] и снова успешное сопоставление со вторым вариантом и т.д. В конечном итоге переменная **Head** становится равной 4, а **Tail=[]** и теперь после рекурсивного вызова **print_list([])** сопоставление для первого варианта правила завершится успехом – **[]=[]**, этот вариант правила не имеет рекурсии – он является фактом и интерпретируется Прологом как цель достигнута, и целевой запрос считается удовлетворенным.

Пример 2. Рассмотрим Пролог-программу подсчета суммы произведений элементов двух векторов X и Y, представленных списками $X=[x_1,x_2,\ldots x_n]$ и $Y=[y_1,y_2,\ldots y_n]$:

```
domains
  vector=integer*
predicates
  prod(vector,vector,integer)
clauses
  prod([],[],0).
  prod([X|Xs],[Y|Ys],S):- prod(Xs,Ys,Sp),
    S=X*Y+Sp.
goal
  prod([1,2,3],[7,8,9],Res),write("Res=",Res).
```

Предикат *prod(...)* является трехместным: первый аргумент – список X, второй – список Y и третий – сумма произведений списков. В разделе **clauses** описаны два варианта правила для этого предиката.

Первый вариант – утверждение о том, что сумма произведений элементов пустых списков равна 0. Второй вариант правила реализован с использованием хвостовой рекурсии. Выражение *S=X*Y+Sp* означает, что текущая сумма равна сумме

произведений текущих элементов вектора и частичной суммы, полученной на предыдущих шагах вычислений.
Схематично действия при выполнении программы представлены в табл. 21.2. Рекурсивное правило описывает отсроченные вычисления – после рекурсивного вызова остается не выполненным хвост правила, копии которого помещаются в стек до тех пор, пока не произойдет сопоставление с первым вариантом правила. И после того, как это произойдет, частичная сумма Sp'' получит значение 0 и все накопленные в стеке хвостовые вычисления будут выполнены в обратном порядке.

Таблица 21.2.

Действия при выполнении программы

Вызов предиката	**Подстановка**	**Хвостовые вычисления**	**Результаты вычислений**
Prod([1,2,3],[7,8,9],Res)	X=1, Y=7, Xs=[2,3], Ys=[8,9], Res=S	S=X*Y+Sp	S=1*7+ +43=50
Prod([2,3],[8,9],Sp)	X'=2, Y'=8, Xs'=[3], Ys'=[9], S=Sp	Sp=X'*Y'+Sp'	Sp=2*8+ +27=43
Prod([3],[9],Sp')	X''=3,Y''=9, Xs''=[], Ys''=[], Sp=Sp'	Sp'=X''*Y''+Sp''	Sp'=3*9+ +0=27
Prod([],[],Sp'')	Xs''=[], Ys''=[], Sp''=0		Sp''=0

Ниже приведен текст Пролог-программы, в которой реализованы основные операции над списками целых чисел: создание списка, добавление элементов в список, распечатка содержимого списка, удаление элемента из списка, упорядочивание элементов в списке. В программе реализован оконный интерфейс и используется модель общения типа выбора из меню.

```
domains
  Sp1,Sp = integer*
  P,U,Z,Z1,N1,N,N2,X,F,P1,Q  = integer
predicates
    edit_list          window(Sp,P)      view_list(Sp)
    make_list(Sp)    delete_list(Sp)
    process(X,Sp,P)   insert_sort(Sp,Sp)
    attent_window    create(N,Sp,Z)
    list(Sp,integer)    insert(integer,Sp,Sp) inversion(Sp,Sp)
    split(integer,integer,Sp,Sp,Sp)      connect(Sp,Sp,Sp)
    add_list(Sp)  goto(Sp,integer)      prt(char,Sp)
    qrt(char,Sp)  delete(integer,Sp)    sound1  sound2
goal
    edit_list.
clauses
/*-------------------------ОСНОВНОЕ МЕНЮ-------------------------*/
edit_list:-window([],0).
```

```
window(Sp,P):-makewindow(1,31,7,"МЕНЮ РЕДАКТОРА",0,0,25,80),cursor(5,15),
   write("  СОЗДАНИЕ СПИСКА                          - 1 "),cursor(7,15),
   write("  ПРОСМОТР СОДЕРЖИМОГО СПИСКА             - 2 "),cursor(9,15),
   write("  КОНКАТЕНАЦИЯ СПИСКА И ЭЛЕМЕНТА          - 3 "),cursor(11,15),
   write("  УДАЛЕНИЕ                                - 4 "),cursor(13,15),
   write("  УПОРЯДОЧИВАНИЕ СПИСКА ПО ВОЗВРАСТАНИЮ - 5 "),cursor(15,15),
   write("  ВЫХОД ИЗ РЕДАКТОРА                      - 6 "),cursor(20,10),
   write("Введите номер пункта меню: "),sound1,
        readint(X),sound(8,2000),X<8,process(X,Sp,P).
 process(1,Sp,0):-make_list(Sp).
 process(1,Sp,1):-make_list([]).
 process(4,Sp,0):-attent_window,window(Sp,0).
 process(4,Sp,1):-delete_list(Sp).
 process(2,Sp,0):-attent_window,window(Sp,0).
 process(2,Sp,1):-view_list(Sp).
 process(3,Sp,0):-add_list(Sp).
 process(3,Sp,1):-add_list(Sp).
 process(5,Sp,0):-attent_window,window(Sp,0).
 process(5,Sp,1):-insert_sort(Sp,Sp1),view_list(Sp1).
 process(6,Sp,1):-sound2,exit.
 process(6,Sp,0):-sound2,exit.

/*---------------------------СОЗДАНИЕ СПИСКА-------------------------*/
make_list(Sp):-makewindow(2,48,7,"СОЗДАНИЕ
СПИСКА",5,5,15,70),gotowindow(2),cursor(3,8),
           write("Количество       элементов       в       списке       -
"),readint(N),sound(8,2000),cursor(5,8),
           write("Введите элементы списка:"),Z=1,Sp=[], create(N,Sp,Z).
create(0,Sp,Z):-inversion(Sp,[]).
create(N,Sp,Z):-write("#",Z,"-
"),readint(U),sound(8,3000),scroll(1,0),cursor(5,32),N1=N-1,
           Z1=Z+1,create(N1,[U|Sp],Z1).
inversion([],Sp1):-window(Sp1,1).
inversion([H|T],Sp):-inversion(T,[H|Sp]).

/*-------------------------ОКНО ПРЕДУПРЕЖДЕНИЯ----------------------*/
 attent_window:-
makewindow(3,64,7,"ВНИМАНИЕ",10,25,5,30),gotowindow(3),nl,
               write("          Ваш     список     пустой               !
"),sound(50,1000),readchar(L).

/*-----------------------------УДАЛЕНИЕ-------------------------------*/
delete_list(Sp):-makewindow(8,48,7,"УДАЛЕНИЕ",5,15,15,45),gotowindow(8),
           write("                                          "),nl,
           write("      СПИСКА                 -   1       "),nl,
           write("      ЭЛЕМЕНТА В СПИСКЕ      -   2       "),nl,
           cursor(7,5),readint(N),delete(N,Sp).
delete(1,Sp):-makewindow(9,64,7,"",10,23,5,30),gotowindow(9),nl,
           write("                                 СПИСОК      УДАЛЕН      !
"),sound1,readchar(U),window([],0).
delete(2,Sp):-makewindow(4,48,7,"УДАЛЕНИЕ
ЭЛЕМЕНТА",10,15,5,45),gotowindow(4),nl,
           write("Введите номер удаляемого элемента: "),readint(S),N=0,
           split(S,N,Sp,Sp1,[H|T]),connect(Sp1,T,Sp3),view_list(Sp3).
 split(S,N,[],[],[]).
 split(S,N,[H|T],[H|L1],L2):-N1=N+1,N1<S,split(S,N1,T,L1,L2).
 split(S,N,[H|T],L1,[H|L2]):-N1=N+1,split(S,N1,T,L1,L2),N1>=S.
 connect([],L,L).
 connect([N|L1],L2,[N|L3]):-connect(L1,L2,L3).

/*----------------------ПРОСМОТР СОДЕРЖИМОГО СПИСКА------------------*/
view_list(Sp):-makewindow(5,48,7,"СОДЕРЖИМОЕ
СПИСКА",5,15,15,45),gotowindow(5),
           cursor(10,5),list(Sp,1),readchar(L),window(Sp,1).
list([],N).
list(T,7):-readchar(L),scroll(1,0),cursor(10,5),list(T,1).
```

```
list([H|T],N):-
N1=N+1,N2=N1*6,write(H),sound(10,3000),cursor(10,N2),list(T,N1),!.

/*----------------УПОРЯДОЧИВАНИЕ ЭЛЕМЕНТОВ ПО ВОЗРАСТАНИЮ------------*/
insert_sort([],[]).
insert_sort([X|Tail],Sorted_list):-insert_sort(Tail,Sorted_Tail),
          insert(X,Sorted_Tail,Sorted_list).
insert(X,[Y|Sorted_list],[Y|Sorted_list1]):-X>Y,!,
          insert(X,Sorted_list,Sorted_list1).
insert(X,Sorted_list,[X|Sorted_list]).

/*----------------------ДОБАВЛЕНИЕ ЭЛЕМЕНТА К СПИСКУ-----------------*/
add_list(Sp):-makewindow(6,48,7,"ДОБАВЛЕНИЕ                              К
СПИСКУ",5,15,15,45),gotowindow(6),
          cursor(3,5),write("В НАЧАЛО СПИСКА – 1"),cursor(5,5),
          write("В КОНЕЦ СПИСКА  – 2"),cursor(7,5),
          readint(L),sound(8,2000),L<3,goto(Sp,L).
goto(Sp,1):-makewindow(7,48,7,"ДОБАВЛЕНИЕ                                К
СПИСКУ",5,15,15,45),gotowindow(7),
          cursor(5,5),write("ВВЕДИТЕ НОВЫЙ ЭЛЕМЕНТ: "),cursor(5,28),
          readint(X),sound(8,3000),connect([X],Sp,Sp1),cursor(7,5),
          write("ЕЩЕ         –         ?         <         y/n         >
"),readchar(Z),sound(8,2000),qrt(Z,Sp1).
goto(Sp,2):-makewindow(8,48,7,"ДОБАВЛЕНИЕ                                К
СПИСКУ",5,15,15,45),gotowindow(8),
          cursor(5,5),write("ВВЕДИТЕ НОВЫЙ ЭЛЕМЕНТ: "),cursor(5,28),
          readint(X),sound(8,3000),connect(Sp,[X],Sp1),cursor(7,5),
          write("ЕЩЕ         –         ?         <         y/n         >
"),readchar(Z),sound(8,2000),prt(Z,Sp1).
prt('y',Sp1):-goto(Sp1,2).
prt('n',Sp1):-view_list(Sp1).
qrt('y',Sp1):-goto(Sp1,1).
qrt('n',Sp1):-view_list(Sp1).
/*-----------------------------------------------------------------*/
sound1:-sound(15,1000),sound(15,1500),sound(15,2000).
sound2:-sound(15,2000),sound(15,1500),sound(15,1000).
```

21.6. Стандартные предикаты

Стандартные (встроенные в систему) предикаты выполняют разнообразные функции по выполнению ввода-вывода переменных различных типов, работе с файлами, выполнению функций DOS, обработке строк, поддержке графического режима, обеспечение интерфейса с другими системами программирования (C, Паскаль) и т.д. Всего в системе содержится более 200 стандартных предикатов.

Для использования стандартных предикатов необходимо знать имя предиката, способы его вызова, число и типы значений параметров, а также режимы их передачи. Режим передачи описывается обычно в виде шаблона, который показывает, какие из параметров являются *входными* (в шаблоне обозначается **i** или **вх.**), а какие *выходными* (в шаблоне обозначается **o** или **вых.**) для предиката. Приведем основные стандартные предикаты Турбо-Пролога в следующем формате:

предикат(список аргументов) (типы доменов):(øàáëîí)

Каждый предикат дополнен кратким описанием его назначения.

makewindow(НомОкна,АтрЭкр,АтрРамки,СтрРамки,Строка,Столбец,Высота,Ширина,
ЧиститьОкно,ПозСтрРамки,ЗнакиГраницы)
(integer,integer,integer,string,integer,integer,integer,integer,integer,i
nteger,string):(вх,вх,вх,вх,вх,вх,вх,вх,вх,вх,вх),(вых,вых,вых,вых,вых,вы
х,вых,вых,вых,вых,вых)
Создает окно с номером *НомОкна*. Остальные аргументы предиката имеют следующий смысл:
АтрЭкр определяет цвет символа и фона.

АтрРамки – при отличном от 0 значении рисуется граница – линия обрамления окна.
СтрРамки указывает строку, помещаемую в центре верхней границы контура окна.
Строка, Столбец – вертикальная и горизонтальная координаты верхнего левого угла окна.
Высота, Ширина – высота (число строк) и ширина (число колонок) окна.
ЧиститьОкно определяет будет ли чиститься окно после его создания: 0 = Не чистить окно; 1 = Чистить окно.
ПозСтрРамки определяет место размещения заголовока окна (внутри верхней линии рамки окна): -1 = Заголовок в центре; N = Размещает заголовок с указанной позиции.
ЗнакиГраницы описывают, как рисовать рамку окна; этот аргумент состоит строго из шести символов, которые обозначающих: верхний левый угол, верхний правый угол, нижний левый угол, нижний правый угол, горизонтальную линию и вертикальную линию соответственно. Например: "\218|191\192\217\196\179" – граница из одной линии; "\201\187\200\188\205\186" -граница из двух линий; "++++|-" -другой вариант описания границы.

readchar(СимволПеременной)(char):(вых)
Читает единственный символ с текущего устройства ввода, которым является по умолчанию клавиатура, пока оно не будет изменено с помощью *readdevice*.

readdevice(СимволичИмяФайла) (symbol):(вх), (вых)
Устанавливает или выдает текущее устройство ввода
(вх): назначает текущее устройство ввода на открытый файл с данным *СимволичИмяФайла*. Открываемый файл может быть один из стандартных файлов или любой файл пользователя с символическим именем, открытый для чтения или модификации.
(вых): связывает *СимволичИмяФайла* с именем текущего устройства ввода.
Стандартные файлы, которые могут быть открыты для ввода: **com1** – чтение из последовательного порта связи; **keyboard** – чтение с клавиатуры (по умолчанию); **stdin** – чтение из стандартного ввода DOS.

readint(ПеременнаяЦел)(integer):(вых)
Читает целое число с текущего устройства ввода. Преобразование символов не осуществляется, пока *readint* не встретит символ возврата каретки (ASCII 13).

readln(ПеременнаяСтр)(string):(вых)
Считывает строку символов с текущего устройства ввода до символа символ возврата каретки (ASCII 13). Самая большая строка, которая может быть прочитана на экране, 147 символов.

readreal(ПеременнаяВещ)(real):(вых)
Читает вещественное число с текущего устройства ввода, пока не прочитает символ возврата каретки. Если нажат Esc (ключ), *readreal* немедленно не согласуется. *readreal* также не согласуется, если символы не образуют правильное вещественное число или они определяют вещественное число, превышающее допустимые границы.

write(e1,e2,e3, ...,eN):(вх,вх,вх, ...,вх)
Выводит значения констант или переменных в текущее окно или на текущее устройство вывода. *write* может быть связан с произвольным ненулевым числом аргументов ei. Аргументы не могут быть свободными переменными.

writedevice(СимвИмяФайла)(symbol):(вх),(вых)
Устанавливает или выдает текущее устройство вывода.
(вх): переназначает текущее устройство вывода в открытый файл с данным *СимвИмяФайла*. Открытый файл может быть одним из стандартных символических файлов или любым пользовательским файлом, открытым для записи или для модификации.

(вых): Связывает *СимвИмяФайла* с текущим устройством вывода.
Для вывода могут использоваться следующие встроенные файлы: **com1** – последовательный порт; **printer** – параллельный порт принтера; **screen** – экран монитора; **stdout** – стандартный выход DOS; **stderr** – файл стандартных ошибок.

removewindow/0
Удаляет текущее окно и переходит к предшествующему окну.

writef(ФорматСтр,Арг1,Арг2,Арг3, ...):(вх,вх, ...,вх)
Выполняет форматированный вывод. Аргументы Арг1 – АргN могут быть константами или переменными. Форматы задаются в виде строки *ФорматСтр* обычного текста, где символы % отмечают положение аргументов строки. Допустимые спецификации формата содержат обычные символы, которые печатаются без модификации, и формат спецификаций формы %-m.pf. Формат спецификаций означает: - (дефис) показывает, что поля выравниваются слева; m поле десятичное число, описывающее минимальный размер поля; .p поле описывает или точное представление числа с плавающей точкой, или максимальное количество напечатанных в строке символов.
f поле описывает следующие форматы: f – Формат вещественного в фиксированной десятичной системе счисления (такой, как 123.4 или 0.004321); e – формат вещественного в экспоненциальной форме представления; g – формат вещественного в коротком формате (используется по умолчанию); d – формат символов или целых десятичных чисел; u – формат символов или целых чисел как десятичное число без знака; x – формат символов или целых чисел как шестнадцатеричного числа; c – формат символов или целых чисел как символа; R – использует аргумент как ссылку на номер указателя базы данных (только ref домен);
X – использует аргумент как длинное шестнадцатеричное число (строки, номер указателя базы данных); s – формат как строка (символов и строк).

asserta(<факт>)(dbasedom):(вх)
Заносит факт (утверждение) в начало резидентной базы данных (домен, обозначенный как *dbasedom*, автоматически объявляется для каждого предиката из раздела *database*).

assertz(X)(dbasedom):(вх)
Заносит факт (утверждение) X в конец резидентной базы данных.

retract(X):(вх)
В базе данных осуществляется поиск утверждения, голова и тело которого сопоставляются с термом X. Первое такое утверждение удаляется из базы данных. Аргумент (терм X) должен быть конкретизирован составным термом.

retractall(X):(вх)
Удаляет из базы данных все утверждения, функтор и арность которых сопоставимы с X.

frontchar(Стр,ПерСимв,ОстСтр)(string,char,string):(вх,вых,вых),(вх,вх,вых),(вх,вых,вх),(вх,вх,вх),(вых,вх,вх)
(вх,вых,вых): присваивает первый символ строки *Стр* переменной *ПерСимв*, а остаток строки – переменной *ОстСтр*. Возможны и другие комбинации входных и выходных аргументов.

frontstr(ЧислСимв,Стр1,НачСтр,Стр2)(integer,string,string,string):(вх,вх,вых,вых): присваивает первые *ЧислСимв* символов строки *Стр1* переменной *НачСтр*, а остаток строки – переменной *Стр2*.

fronttoken(Стр,Знак,ОстСтр)(string,string,string):(вх,вых,вых),(вх,вх,вых),(вх,вых,вх),(вх,вх,вх),(вых,вх,вх)
(вх,вых,вых):переменной *Стр* присваивается результат конкатенации *Знак* и *ОстСтр*. *Знак* может быть группой символов, задающих допустимое имя, либо символьным представлением числа, либо одиночным символом, отличным от

```
пробела. В други комбинациях входных и выходных аргументов должны быть
означены минимум два аргумента предиката.

cursor(Стр,Кол)(integer,integer):(вх,вх),(вых,вых)
(вх,вх):помещает курсор в позицию с координатами (Стр,Кол).
```

22. РЕШАТЕЛЬ ВЫЧИСЛИТЕЛЬНЫХ ЗАДАЧ

Решатели вычислительных задач принадлежат к классу интеллектуальных пакетов прикладных программ (ИППП). Они позволяют пользователю решать задачи с использованием ЭВМ по описанию и исходным данным без программирования алгоритма решения задачи – программирование осуществляется автоматически программой планировщиком из набора готовых программных модулей, относящихся к конкретной предметной области. Мы рассмотрим систему TK Solver, являющуюся характерным представителем решателей вычислительных задач.

22.1. Вычислительные модели и задачи, синтез программ

В общем случае ИППП включает в свой состав следующие компоненты: подсистему общения пользователя с пакетом, подсистему планирования решений задач (планировщик) и базу знаний о программных модулях, составляющих функциональное наполнение пакета.

Планировщик является *ядром ИППП*, основная цель его – получение плана решения задачи пользователя, формулировка которой поступает на вход подсистемы планирования. План решения задачи в ИППП трактуется как поиск программы, решающей эту задачу. При этом программа решения задачи может конструироваться либо явно на каком-либо из алгоритмических языков программирования, либо существует в виде шагов работы планировщика. В первом случае ИППП является *транслятором* с декларативного языка описания задач на алгоритмический язык программирования. Во втором варианте, когда программа решения задачи не конструируется явно, а существует в виде шагов работы планировщика, планировщик работает в режиме интерпретации и является, по сути, *интерпретатором* с языка описания задач.

Как в первом, так и во втором вариантах работа планировщика отличается от функционирования обычного транслятора с алгоритмического языка программирования. Если функцией обычного транслятора является преобразование входной программы с входного алгоритмического языка программирования в язык машинных команд ЭВМ, то основная задача планировщика – *синтез алгоритма* программы решения задачи по декларативному описанию формулировки задачи. Синтез осуществляется с использованием *базы знаний* (БЗ). Типовая структура системы синтеза программ на основе представления знаний приведена на рис. 22.1.

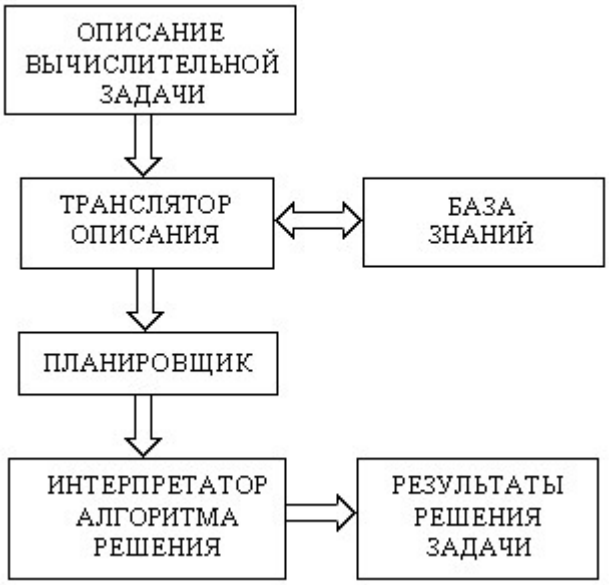


Рис. 22.1. Структура системы синтеза программ

База знаний в ИППП описывается на языке спецификаций, задачи формулируются на языке формулировок задач. В общем случае это может быть один и тот же язык. Пользователь, использующий ИППП для решения прикладных задач, выступает обычно в двух ролях:

- как инженер по знаниям, т.е. специалист в предметной области, когда описывает содержимое базы знаний ;
- как кодировщик задач, когда формулирует задачи с использованием накопленных в ИППП знаний.

Знания в БЗ могут быть представлены с использованием различных моделей. В наиболее развитых ИППП в качестве модели предметной области используются так называемые *вычислительные модели*. На основе вычислительных моделей формулируются *вычислительные задачи*.

Вычислительная задача формулируется в рамках некоторой предметной области, рассматриваемой как совокупность объектов и отношений между ними. При этом рассматриваются только такие объекты, на основе которых можно осуществлять вычисления. Между объектами существуют отношения вычислимости, задаваемые в виде *функциональных отображений*. Объекты могут быть *простыми* или *составными*. В составном объекте между его компонентами имеют место структурные отношения.

Объекты и отношения между ними описываются на некотором языке. Объектам в этом языке соответствуют переменные, а отношениям – функции. Значениями переменных являются данные.

Вычислительная задача имеет следующую форму:

ЗНАЯ M ВЫЧИСЛИТЬ Y1,Y2,...,YN ПО X1,X2,...,XM.

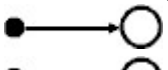
Здесь **M**, **Y1,Y2,...,YN**, **X1,X2,...,XM** – переменные, которые имеют смысл, определяемый их вхождением в задачу. Идентификаторы **ЗНАЯ, ВЫЧИСЛИТЬ** и **ПО** имеют фиксированный смысл и служат для разделения переменных. Переменные **X1,X2,...,XM** являются *входными* для задачи, значения их задаются в постановке задачи. Переменные **Y1,Y2,...,YN** – *выходные*, значения их требуется

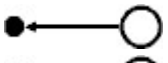
вычислить. **M** – переменная, значение которой выражает условия задачи. Данные, являющиеся значением **M**, выражают знания в виде *вычислительных моделей*, включающих в свой состав переменные и отношения между ними.

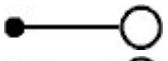
Вычислительную модель (ВМ) можно представлять *графически*, с помощью языка спецификации и с *помощью формул* некоторого логического языка.

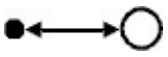
Графическое представление вычислительной модели – это семантическая сеть специального вида, узлам которой соответствуют переменные и отношения. Переменные обозначаются точками, отношения кружочками, овалами или прямоугольниками, внутри которых помещается описание реализации отношения.

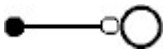
Связи между переменными и отношениями могут быть следующих типов:

 – входная (переменная является входной для отношения).

 – выходная (переменная является выходной для отношения).

 – слабая связь (переменная может быть как входной, так и выходной).

 – сильная связь (значение переменной меняется отношением).

 – определяющая связь (значение переменной определяет возможность применения отношения).

На рис. 22.2 приведен пример вычислительной модели КВАДРАТ, где в качестве отношений используются отношения типа уравнение. Все переменные, входящие в уравнения, являются *слабосвязанными*, т. е. любая из переменных может быть либо входной, либо выходной для уравнения.

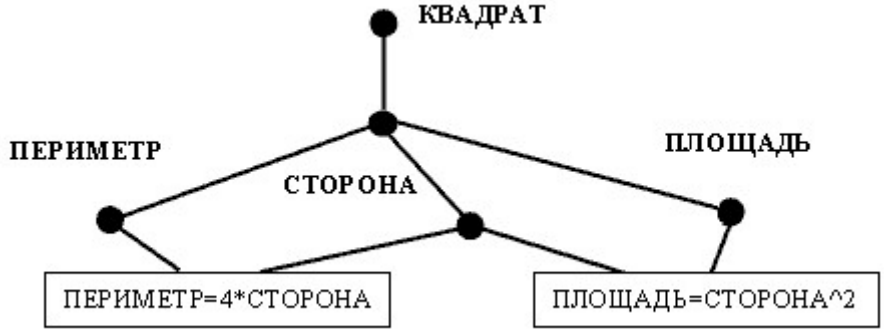


Рис. 22.2. ВМ КВАДРАТ с отношениями типа уравнение

На одной вычислительной модели может быть решено множество вычислительных задач. Так, на приведенной на рис. 22.2 ВМ можно решить следующие задачи:

вычислить ПЕРИМЕТР по СТОРОНА;
вычислить СТОРОНА по ПЕРИМЕТР;
вычислить ПЛОЩАДЬ по СТОРОНА;
вычислить СТОРОНА по ПЛОЩАДЬ;
вычислить ПЕРИМЕТР по ПЛОЩАДЬ;
вычислить ПЛОЩАДЬ по ПЕРИМЕТР.

Проводить такие вычисления возможно при условии, что для уравнения всегда можно построить его разрешение относительно любой из входящих в него переменных. Если, например, в уравнение входит ***n*** переменных, то будем иметь ***n*** функций разрешения. Записывается это обычно в виде:

$$x_j = f_i^j(x_1, x_2, ..., x_{j-1}, x_{j+1}, ... x_n)$$

Здесь X_j – переменная, входящая в уравнение, f_i^j – реализация функции, вычисляющей значение j-й переменной в уравнении с номером i ($i=1,...,k$; k – число уравнений в ВМ). Уравнения считаются независимыми друг от друга и являются компактным описанием множества разрешающих функций.

Входной язык ИППП обычно допускает использование уравнений при описании вычислительных моделей. После преобразования такого описания во внутреннее представление базы знаний уравнения заменяются функциями разрешения и в базе знаний уравнения хранятся в форме одно-операторных отношений, т. е. без использования слабосвязанных переменных. Для уравнений из примера на рис. 22.2 такими разрешениями являются следующие:

Периметр = f_1^1 **(Сторона);**

Сторона = f_1^2 **(Периметр);**

Площадь = f_2^1 **(Сторона);**

Сторона = f_2^2 **(Площадь).**

В *одно-операторном отношении* имеется одна выходная переменная, остальные являются входными для отношения и одно-операторное отношение показывает, значения каких переменных должны быть известны, для того, чтобы можно было вычислить значение выходной переменной. Замена уравнений в ВМ их функциями разрешения позволяет избавиться от слабосвязанных переменных и тем самым получить описание вычислительной модели в виде двудольного ориентированного графа (орграфа). На рис. 22.3 приведено графическое представление ВМ КВАДРАТ, в которой такая замена выполнена, т.е. вместо отношений типа уравнение используются представляющие их *функции разрешения* в виде *одно-операторных отношений*.

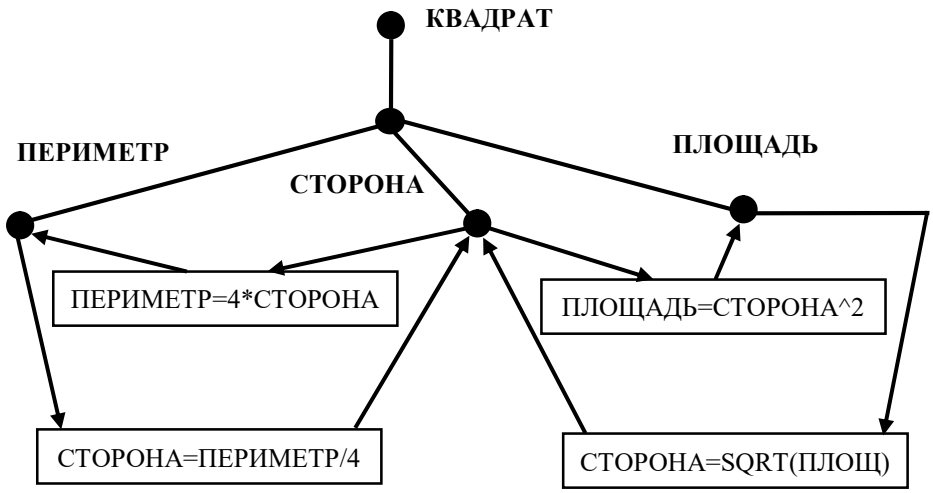


Рис. 22.3. ВМ КВАДРАТ с одно-операторными отношениями

Однако ВМ с использованием одно-операторных отношений еще не является алгоритмическим описанием – в модели не зафиксирован порядок выполнения операторов. Поэтому такое описание, по сути, задает потенциальное множество алгоритмов, которые могут быть сконструированы с использованием отношений

вычислительной модели. Конкретный алгоритм может быть получен, если к ВМ присоединить формулировку задачи в виде:

ЗНАЯ <***ВЫЧИСЛИТЕЛЬНУЮ МОДЕЛЬ*** >
ВЫЧИСЛИТЬ < ***ВЫХОДЫ ЗАДАЧИ*** > **ПО** < ***ВХОДАМ ЗАДАЧИ*** >.

Вычислительная модель и присоединенная к ней формулировка задачи образуют ***модель задачи***. Формулировка задачи разбивает множество входящих в вычислительную модель переменных на два подмножества: *входные переменные*, значения которых известны до решения задачи, и *выходные переменные*, значения которых необходимо вычислить.

Если входных переменных достаточно для нахождения выходных с использованием разрешающих функций, которые содержит вычислительная модель, то задача является *разрешимой*, и для ее решения может быть сконструирована программа.

Обычно поиск решения задачи (т.е. программы) сводится к отысканию пути в орграфе модели задачи, ведущего от исходных данных задачи к ее результатам. Нахождение такого пути в ИППП осуществляет *планировщик*. Планирование при этом называют планированием от данных, возможен и другой подход: планирование от цели, в этом варианте планировщик ищет путь от выходных переменных задачи к входным. Возможна и комбинация этих вариантов (например, в системе ПРИЗ используется метод прямой и обратной волны). Выписывание встречающихся на этом пути функций разрешения в форме, например, операторов присваивания, дает описание плана (алгоритма) решения задачи. Так, например, если на ВМ КВАДРАТ сформулировать задачу:

ЗНАЯ ***КВАДРАТ*** **ВЫЧИСЛИТЬ** ***ПЛОЩАДЬ*** **ПО** ***ПЕРИМЕТР***, будет получен следующий план вычислений в форме присваиваний:

Сторона:= f_1^2 **(Периметр).**

Площадь:= f_2^1 **(Сторона);**

В функциональной форме это записывается в следующем виде:

Площадь = f_2^1 **(**f_1^2 **(Периметр)).**

Графическое представление вычислительной модели данной задачи приведено на рис. 22.4. И если теперь для каждой из входящих в план вычислений функций имеется реализация в виде, например, программного модуля, то выполнение (интерпретация) модулей приведет к получению результата задачи.

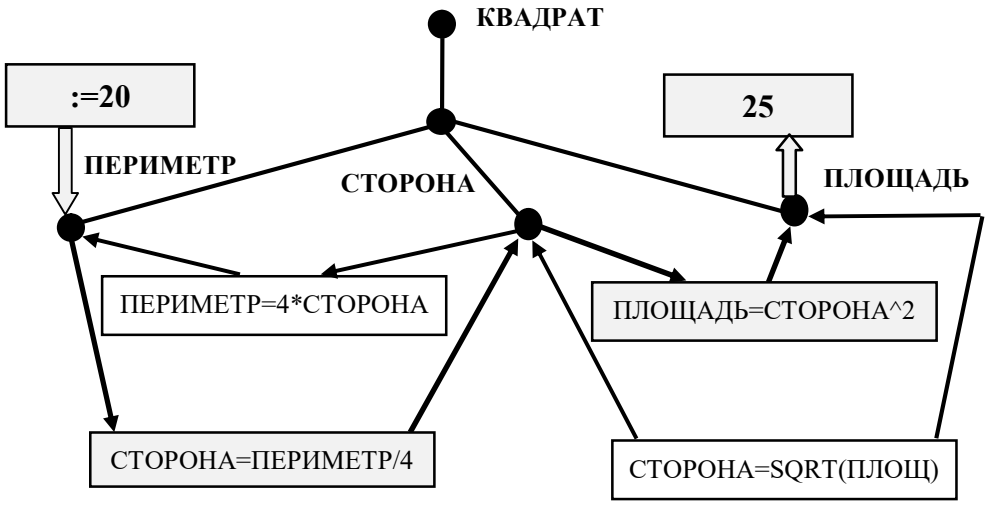


Рис. 22.4. ВМ задачи нахождения площади квадрата по периметру

Графическое представление ВМ достаточно полно описывает существо вычислительных задач. Однако в этом представлении отсутствует информация о типах объектов модели задачи не с позиций проведения вычислений в принципе, а с использованием ЭВМ. Для реализации вычислений на ЭВМ с каждым объектом необходимо связать тип его данного с точки зрения множества принимаемых этим объектом значений (целое, вещественное, символьное и т. д.). Такой информацией ВМ доопределяется с использованием языка спецификации вычислительных задач и построением поддерживающей этот язык программной системы синтеза программ. В настоящее время разработан и используется на практике ряд языков спецификаций вычислительных задач, программы решения которых находятся с применением рассмотренных и более сложных вычислительных моделей (УТОПИСТ, ДЕКАРТ, Язык решения задач системы МИКРОПРИЗ и др.).Устройство языка спецификаций и его применение для описания и решения вычислительных моделей и задач рассмотрим на примере системы ТК Solver.

22.2. Характеристика решателя задач ТК Solver

Программная система **ТК Solver** является представителем универсальных решателей вычислительных задач. Такие решатели задач являются хорошим инструментом в руках пользователей различной квалификации при проведении инженерных, научных и финансовых расчетов.

Система ТК Solver является открытой и позволяет:

- описывать в интерактивном режиме достаточно сложные вычислительные модели на входном языке системы;
- накапливать описания в библиотеке моделей (базе знаний системы);
- конструировать с использованием библиотечных моделей другие описания задач;
- автоматически генерировать и выполнять программы решения этих задач.

Интерфейс системы ТК Solver

Рассматриваемая нами версия системы ТК Solver 1.1 реализована в виде интегрированной среды и имеет стандартный интерфейс приложений Windows. Работа с системой осуществляется с помощью команд основного меню и кнопок панели инструментов, позволяющих удобно задавать наиболее часто употребляемые команды.

Основное меню системы ТК Solver содержит следующие меню: *File (Файл), Edit (Правка), Commands (Команды), Format (Формат), Window (Окно), Application (Приложение), Help (?).* Рассмотрим краткое описание основных команд меню.

Меню **File** содержит следующие команды по работе с файлами моделей задач.

New (Новый) – создание нового файла описания модели задачи;

Open (Открыть) – открытие существующего файла и загрузка его содержимого в качестве текущей модели задачи;

Merge (Вставить) – активизирует диалог загрузки файла. Содержимое файла обновляет текущую модель задачи, т. е. если во включаемом файле есть описания переменных, графиков, списков и т. д., совпадающие с такими же описаниями текущей модели задачи, то они заменяют текущие. Уравнения из файла полностью переносятся в текущую модель, если в модели уже есть такие уравнения, то они помещаются в конец;

Include (Включить) – включение в текущую модель задачи компонентов других моделей из файла. Включаемые компоненты отображаются в текущей модели другим цветом, не могут быть изменены и не попадают в выходной файл после его сохранения по команде *Save*;

Save (Сохранить) – сохранение всех панелей текущей модели задачи в файле под тем же именем;

Save As (Сохранить как) – то же, что и *Save*, но в файле с другим именем;

Save Window (Сохранить окно) – сохранение активного окна (панели) в файле;

Import (Импорт) – загрузка списков в текущую модель задачи из файла, созданного ранее по команде Export;

Export (Экспорт) – сохранение списков текущей модели задачи в файле в формате **ASCII** (расширение имени файла **.asc**);

Print (Печать) – вывод на печать содержимого панелей текущей модели задачи;

Print Preview (Предварительный просмотр) – просмотр перед печатью выводимой информации;

Print Setup (Параметры печати) – установка параметров для печати;

Recent Files (Недавние файлы) – список файлов;

Exit (Выход) – выход из системы.

Меню **Edit (Правка)** содержит команды редактирования панелей текущей модели задачи.

Меню **Commands (Команды)** обеспечивает различные режимы решения задачи на текущей вычислительной модели, содержит следующие команды:

Display Plot (Отобразить график) – активизировать процесс построения графиков, описанных на панели **Plot Sheet,** на экран выводится выбранный на этой панели график;

Solve (Решить) – выполнить вычисления в режиме прямого решателя;

List Solve (Решить список) – выполнить списковые вычисления;

Block Solve (Решить блок) – выполнить списковые вычисления для элементов списков, указанных в команде;

Abort Operation (Прервать операцию) – прервать выполнение операции;

Examine (Исследовать) – выполнить вычисления для заданных в команде функций, переменных или введенного здесь же выражения;

List Fill (Заполнить список) – генерировать числовые значения элементов выбранного списка;

Put Values To Lists (Поместить значения в список) – присвоить содержимое поля Input (или Output) панели ***Variable Sheet*** элементу списка, связанного с этой переменной, номер элемента указывается в команде;
Get Values From Lists (Получить значения из списка) – вывести на панель ***Variable Sheet*** в поле Input (Ввод) содержимое элемента списка, связанного с выбранной переменной;
Display Solution Time (Отобразить время решения) – показать время решения текущей задачи.
Меню **Format** содержит диалоговые команды для установки параметров системы (команда *Settings)*, подключения и изменения применяемых на отдельных или всех панелях шрифтов (команда *Set Font,* символы кириллицы можно использовать в качестве комментариев), изменения цветовой палитры панелей (команда *Set Color)*.
Меню **Window** (Окно) содержит команды для управления отображением и размещением панелей.
Меню **Applications** (Приложения) включает диалоговые команды для обращения к системным библиотекам моделей, любая из моделей, хранимых в виде файлов, может быть включена в текущую модель задачи и использоваться для вычислений.
Меню **Help** (?) содержит команды для получения справочной информации.
Вычислительная модель задачи в решателе ТК Solver описывается на нескольких разнотипных взаимосвязанных панелях, заполняются панели в интерактивном режиме с помощью встроенных в систему средств редактирования и сохраняются в рамках единого файла с расширением **.TKW**.
Панели размещены внутри стандартных окон Windows, с которыми можно выполнять операции оконного интерфейса: раскрывать, перемещать, изменять размеры, сворачивать и т. д. Панели имеют форму таблиц, элементы которых заполняются соответствующими значениями с учетом назначения панели (для описания переменных, для описания функций и т.д.) и типа поля панели.
В ТК Solver вычислительная задача описывается с использованием следующих основных панелей:

Variable Sheet – переменных модели;
Rule Sheet – предложений вычислимости постановки задачи*;*
Function Sheet – заголовков функций и процедур;
Unit Sheet – единиц измерения переменных вычислительной модели;
List Sheet – списков, используемых в модели;
Plot Sheet – графиков;
Table Sheet – таблиц;
Format Sheet – форматов данных переменных модели;
Comment Sheet – комментариев.

Содержимое любого поля панели активизируется с помощью курсора, а раскрывается с помощью правой кнопки мыши. При раскрытии текущего поля панели осуществляется переход к другой, связанной с этим полем панели.

Основы решения вычислительных задач

Рассмотрим принципы описания и решения простейших вычислительных задач в системе ТК. Для того чтобы в ТК появилась ***текущая вычислительная задача,*** необходимо ее описать как минимум на двух панелях:

- на панели *Rule Sheet* описать вычислительную модель в виде условных и безусловных предложений вычислимости;

- на панели *Variable Sheet* сформулировать задачу путем явного разбиения переменных входящих в ВМ на *входные* (типа Input) и *выходные* (типа Output) и для входных переменных задать начальные значения.

После такого описания в ТК Solver появляется ***текущая вычислительная модель задачи.***

Вычислительная модель представляет собой *декларативное описание отношений* вычислимости на *языке спецификаций вычислительных задач*. В рамках вычислительной модели можно обращаться к функциям, описывающим алгоритмы с использованием операторов бейсикоподобного алгоритмического языка.

Для получения решения задачи на текущей модели задачи необходимо выполнить команду Commands | Solve (Команды | Решить) или нажать клавишу <F9>. На рис. 22.5 показано содержимое панелей ***Rule Sheet*** и ***Variable Sheet*** **в окне** ТК Solver после решения задачи с формулировкой:

ЗНАЯ ***Текущую модель задачи*** **ВЫЧИСЛИТЬ** ***S*** **ПО** ***P***

Здесь переменным с именами ***S*** и ***P*** в графическом представлении модели задачи (рис. 22.4) соответствуют объекты ПЛОЩАДЬ и ПЕРИМЕТР соответственно.

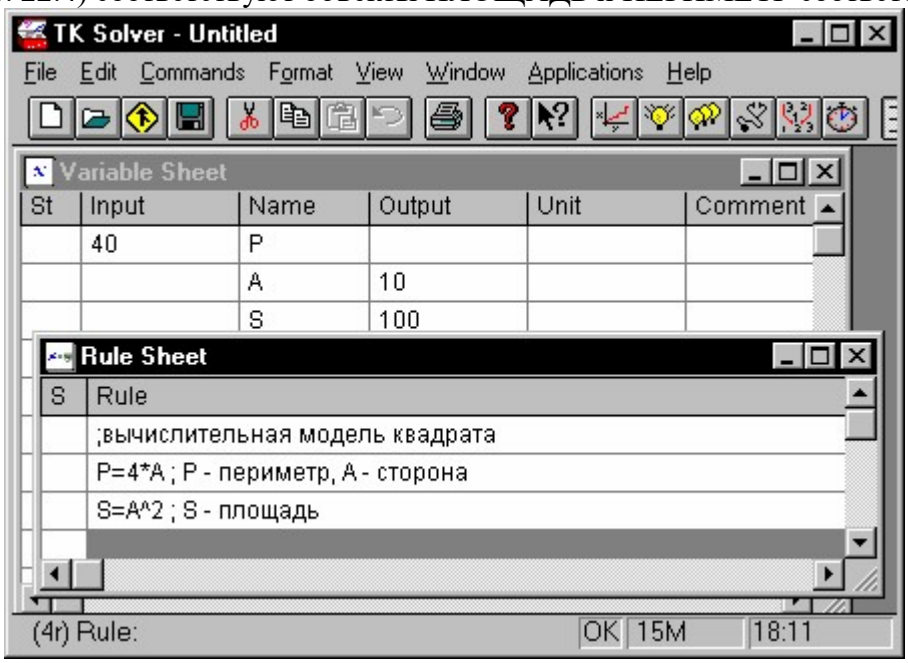


Рис. 22.5. Вид панелей переменных и правил в окне ТК Solver

Содержимое всех панелей текущей модели задачи можно сохранить в файле с расширением TKW. В дальнейшем этот файл можно загрузить и использовать для решения других задач.

Решение задач на вычислительной модели, описанной в ТК Solver на панели ***Rule Sheet***, может осуществляться в трех режимах:

- в режиме ***прямого решателя,*** который активизируется с помощью команды Commands | Solve (Команды | Решить) или нажатием клавиши <F9>;
- в режиме ***спискового решения***, который активизируется командой Commands | List Solve (Команды | Решить списком или нажатием клавиши F10;
- в режиме ***итеративного решателя***, который активизируется автоматически из других режимов, если возникает прерывание по ошибке из-за невозможности вычислить значение какой-либо переменной, входящей в уравнение и для этой

переменной назначен атрибут G в поле St на панели описания переменных ***Variable Sheet***.

На рис. 22.5 приведен пример, когда решение получено с использованием режима прямого решателя. ***Прямой решатель*** находит значения всех переменных, которые можно вычислить с использованием предложений вычислимости, заданных на панели ***Rule Sheet*****,** с использованием входных переменных панели ***Variable Sheet***. Если для предложения вычислимости входных данных недостаточно, то оно в вычислениях не участвует, и в поле **S** на панели ***Rule Sheet*** помечается символом *****.

В процессе решения решатель проверяет вычисления на непротиворечивость, т. е. если, например, в процессе решения для какого-либо из уравнений все переменные оказались входными, то проверяется выполнение равенства левой и правой частей уравнения. В случае не выполнения равенства вычисления прекращаются, выводится сообщение об ошибке, а переменные уравнения и само уравнение на соответствующих панелях помечаются символом **>**. Для исключения уравнения из вычислений достаточно перед ним поставить символ **;** (точка с запятой), т. е. объявить его комментарием.

Для выполнения ***списковых вычислений***, т.е. решения одной и той же задачи на различных исходных данных, необходимо предварительно с каждой переменной, участвующей в вычислениях, связать список. Для этого в поле **St** на панели ***Variable Sheet*** указать атрибут **L**, а для входных переменных – атрибут **Input** (в поле **Input** появится **0**, а не первый элемент списка) и в связанные с ними списки занести множество принимаемых этой переменной значений. После этого можно активизировать списковые вычисления нажатием клавиши <F10>.

В процессе списковых вычислений прямой решатель вызывается столько раз, сколько элементов в списках входных переменных. Найденные на каждом цикле значения выходных переменных задачи помещаются в соответствующие списки для выходных переменных. При этом номер элемента в списке является номером решения одной и той же задачи на различных исходных данных. Результаты решения можно просмотреть в соответствующем списке, для этого необходимо активизировать панель ***List Sheet*** и раскрыть список с помощью правой кнопки мыши.

Режим списковых вычислений целесообразно использовать для выяснения характера функциональной зависимости между входными и выходными данными решаемой задачи. Эту зависимость можно в дальнейшем представить либо в виде таблицы, либо в графическом виде в форме графика в декартовой системе координат, столбиковой или круговой диаграммы.

В процессе решения задачи может оказаться, что получение решения невозможно из-за неполноты задания исходных данных задачи. В этом случае целесообразно применять режим ***итеративного решателя***. Для того, чтобы перевести систему ТК Solver в режим итеративного решения уравнений, необходимо неизвестным переменным назначить тип **Guess** в поле **St** на панели ***Variable Sheet*** и в поле **Input** присвоить им начальное приближение.

Рассмотрим два уравнения с двумя неизвестными:

$$y = x*\sin(x)$$

$$x*y = \cos(x + y)$$

Пусть эти уравнения описаны в виде правил. Если переменной x назначить тип **Input** и значение **x=0.5**, то в режиме прямого решателя (после нажатия клавиши **<**F9>) будет выведено сообщение об ошибке, так как после вычисления значения **y** с

помощью первого уравнения прямой решатель подставит значения **x** и **y** во второе уравнение, и равенство для него не будет выполняться.
В случае когда переменной **x** назначен тип **Guess** и значение **x =0.5,** будет активизирован режим итеративного решения, при этом после возникновения ошибки из-за второго уравнения решение не будет прекращено, а произойдет перевычисление значения переменной **x,** вычислено новое значение **y** и цикл вычислений будет повторяться до тех пор, пока будет инициироваться прерывание по ошибке. Окончательное решение для приведенных уравнений **x = .727** и **y = .484**.

22.3. Язык спецификации вычислительных задач

В ТК Solver ***вычислительная модель*** представляет собой совокупность объектов и отношений между ними. Каждый объект в модели принадлежит к некоторому классу (типу данных) и в описании модели в ТК Solver объект соотносится с именем переменной. Тип переменной определяется множеством значений, которые может принимать переменная, и множеством операций над этими значениями. ВМ в ТК описывается в соответствии с правилами синтаксиса и семантики конструкций языка описания вычислительных моделей и задач, в системе он не имеет специального названия, будем его называть Язык Спецификации Вычислительных Задач (ЯСВЗ).
В ВМ ***объект*** каждого типа имеет уникальное имя. В ТК Solver ***имя*** может иметь в длину до **200** символов и включать буквы **A-Z** и **a-z**, цифры **0-9** и специальные символы **@**, **#**, **$**, **%**, и **_.** Имя не может начинаться с цифры, содержать внутри себя пробелы, строчные и прописные буквы в имени различаются, т. е. например, имена **Abc** и **abc** обозначают разные объекты. Для системных имен это ограничение не действует (SIN и Sin обозначают одно и то же).
В ЯСВЗ допускается использовать ***переменные следующих типов***:

- числовые переменные (целые, вещественные и комплексные);
- символьные переменные;
- булевы переменные.

Явные описатели типов в языке отсутствуют, тип переменной определяется либо видом принимаемых переменной значений при их инициализации, либо местом использования переменной, либо указанием определенных атрибутов в полях панелей объявления переменных.
В ЯСВЗ можно использовать и структурированные типы данных. Если в модели задачи требуется проводить вычисления многократно с различными исходными данными, тогда с переменной в ТК Solver можно связать ***список*** – последовательность однотипных значений, называемых *элементами* списка. В свою очередь, списки можно объединять (связывать) в более сложные структуры – *таблицы*, соответствие между элементами двух числовых списков можно представлять в форме геометрической интерпретации в виде *графиков, столбиковых* и *круговых диаграмм*.
Числа представляются в ТК Solver в стандартной или экспоненциальной форме, могут иметь до 16 значащих цифр в диапазоне от 1E-307 до 1E308 (абсолютное значение). Примеры записи чисел: 1.09092, 1843756000000000000000, 6.672E-11. Для записи комплексных чисел используется специальная конструкция, называемая ***парой***.

Символьные значения заключаются в двойные кавычки “, при этом имена переменных, хранящие символьные значения должны начинаться с апострофа ‘.
Примеры символьных переменных и констант: 'x, 'Bank, 'a_row_1, '_line, 'aj%d4, "line d","a*j/d[4]", "K".
Символьные значения, используемые для обращения к спискам или выступающие в качестве аргументов функций для указания объектов ТК Solver (переменные, списки и т.д.), записываются или с апострофом в качестве префикса имени, или заключаются в двойные кавычки. Примеры правильного описания символьных значений: 'Xupper[i], given('ab,'cd,1,0), "Xupper"[i], given("ab","cd",1,0). Примеры неверного описания: "X upper"[i], given("ab,cd",1,0)
Простейшими синтаксическими единицами в ЯСВЗ являются ***выражения***. В ТК Solver применяются следующие выражения: арифметические, логические и символьные выражения.
Арифметические выражения строятся из знаков арифметических операций, имен переменных, обращений к функциям и спискам, числовых констант, круглых скобок. К арифметическим операциям относятся следующие операции: сложение (+), вычитание (-), умножение (*), деление (/) и возведение в степень (^). Примеры арифметических выражений:

1. **a + b / (c * d)**
2. **(b > 3) + a**
3. **(a+b)^ln(x)**
4. **exp(cos(x)+sin(y))**
5. **-SIN(X)**

Логические выражения строятся с использованием знаков операций сравнения, имен переменных, констант, вызовов булевых функций и принимают значения **1** (ИСТИНА) или **0** (ЛОЖЬ). К *операциям сравнения* относятся операции:

- сравнение на равенство (=);
- сравнение на неравенство (<, >, <=, =<, >=, =>, <>, ><).

В ТК Solver реализованы следующие булевы функции:

- NOT(x) – логическое НЕ ;
- EQV(x,y) – эквивалентность;
- AND(<список логических выражений>) – логическое И;
- IMPLY(x,y) – логическая импликация;
- OR(<список логических выражений>) – логическое ИЛИ.

Здесь ***x*** и ***y*** – логические выражения, а ***<список логических выражений>*** - последовательность логических выражений, разделенных запятыми.
К ***символьным выражениям*** относятся:

- переменные символьного типа;
- выражения, в которых используются символьные переменные.

Объекты (переменные) в ТК Solver связываются между собой ***отношениями.*** В моделях допускается применение следующих видов отношений:

- ***реляционные отношения***, задаются путем явного перечисления элементов отношений и представляются в ТК Solver в форме таблиц или (при геометрической интерпретации бинарных таблиц) графиков;
- ***отношения вычислимости***, задаются в функциональной форме в виде предложений вычислимости, которые могут быть *безусловными и условными.*

Безусловные предложения вычислимости

Безусловные предложения вычислимости в ТК Solver задаются в виде уравнений и программных отношений.

Уравнения. Различают арифметические и неарифметические уравнения.

Арифметические уравнения имеют вид:

<AB1>=<AB2>,

где **<AB1>, <AB2>** -арифметические выражения.

Хранятся арифметические уравнения в рамках текущей вычислительной модели в виде одно-операторных отношений, т.е. для каждой переменной конструируется разрешающая (вычисляющая) ее функция. Система ТК Solver в арифметических выражениях позволяет использовать библиотечные функции. При этом система обеспечивает вычисление значений переменных, используемых в качестве аргументов только в тех функциях, которые имеют обратные функции (например, для функции **SIN(X)** имеется обратная **ASIN(X)** и т.д.).

Примеры описания арифметических уравнений:

1. **a + b = c * d**
2. **remainder = mod(a,b)**
3. **theta = pi()^2 + r/2**
4. **y * (cos(x) – sin(y)) = ln(x)**
5. **x = 5**
6. **(x,y) = ptor(r1,i1) + ptor(r2,i2)**

Неарифметические уравнения задаются с использованием символьных выражений, при этом хотя бы в одной из частей уравнения должно быть только одно имя переменной. Примеры описания неарифметических уравнений:

1. **res=”РЕЗУЛЬТАТ”**
2. **“ТАБЛИЦА”=TABL**

Программные отношения представляют собой отношения, которые задаются с помощью функций: ***библиотечных (встроенных)*** или ***пользователя***, последние представляются в ТК Solver с помощью панелей ***Function Subsheet***. Программные отношения описываются в виде обращений к функциям, в обращении указывается имя функции и все ее фактические параметры – переменные модели задачи, которые связаны с данным программным отношением.

Общая форма описания обращения к функции имеет вид:

CALL <ИМЯ ФУНКЦИИ> (<СПИСОК ВХОДНЫХ ПЕРЕМЕННЫХ>; <СПИСОК ВЫХОДНЫХ ПЕРЕМЕННЫХ>)

Здесь список входных переменных отделяется от списка выходных переменных точкой с запятой.

Условные предложения вычислимости

Условные предложения вычислимости задают условия применимости тех или иных отношений и имеют две формы записи. Первая форма имеет вид:

IF <ЛОГ.ВЫРАЖ.> THEN <БЕЗУСЛОВНОЕ ПРЕДЛОЖЕНИЕ>

Данное условное предложение определяет условия, когда связанное с ним отношение *не будет* использоваться в вычислениях, а именно: когда логическое выражение *ложно*. Если же логическое выражение *истинно,* безусловное

предложение вычислимости может использоваться в вычислениях, но это не означает, что это случится обязательно, все зависит от того, понадобятся ли в вычислениях переменные, которые используются в нем.
Другая форма записи уловного предложения вычислимости:

IF<ЛОГ.ВЫРАЖ.> THEN <БЕЗУСЛОВНОЕ ПРЕДЛОЖЕНИЕ1>
ELSE <БЕЗУСЛОВНОЕ ПРЕДЛОЖЕНИЕ2>

Данная форма определяет применение соответствующего безусловного предложения вычислимости, как для истинного, так и для ложного значения логического выражения. Но опять же это необходимые условия применимости, для того, чтобы в вычислениях было использовано первое либо второе предложение вычислимости нужно, чтобы они понадобились в вычислениях и эти вычисления были разрешимыми относительно неизвестных переменных.
Примеры условных предложений вычислимости:

1. IF x>=0 THEN y=sqrt(x) ELSE msg='sq_err
2. IF and(w>0, given('z)) THEN q=ln(w)*z
3.IF evltd('a) THEN b=a*c+d

Функции

В ТК Solver поддерживается использование функций следующих типов:

- ***функции–списки***, задают различные виды соответствий между элементами двух списков;
- ***процедуры-функции***, содержат описания алгоритмов решения подзадач на бейсикоподобном языке программирования и представляют собой множество операторов (условных, безусловных, циклов и др.), выполнение которых определяется индуцируемым в процедуре алгоритмом;
- ***функции-вычислительные модели (ВМ-функции)***, задают описания вычислительных моделей в виде объектов и отношений между ними. При обращении к такой функции в ТК Solver возникает подзадача, заданная на множестве фактических параметров функции, при этом подзадача будет разрешима в том случае, когда множество входных (значения которых известны до вызова функции) переменных позволит вычислить на определенных в теле функции отношениях значения выходных (результатов функции) переменных модели. В свою очередь, в теле функции могут быть обращения к другим ВМ-функциям, а также к процедурам-функциям и функциям-спискам.

В системе ТК Solver имеется ***библиотека***, которая содержит более 100 различных ***встроенных функций*** и ***процедур-функций***: тригонометрические и гиперболические функции, функции для действий над комплексными данными, булевы функции, функции и процедуры для работы со списками, функции для обработки строковых данных, функции для преобразования полярных и прямоугольных координат, функции для работы с файлами. Кроме того, имеются функции вычисления квадратного корня, возведения в степень, определения знака, вычисления ближайшего целого и ряд других. В состав библиотеки входят математические константы: PI() – константа p= 3.141592653589793 и E() – константа e= 2.718281828459045.
Примеры использования в уравнениях встроенных тригонометрических функций:
1. y – y0 = sin(x)/x;

2. a/sind(A) = b/sind(B);
3. exp(-t^2) = phi;
4. circumference = 2*pi()*radius.

Кроме использования ***встроенных функций***, можно описывать и использовать ***функции пользователя***.

22.4. Описание функций пользователя

Решение сколько-нибудь сложной задачи в предметной области решателя требует ее сведения к решению такой совокупности подзадач, решение которых известно. Для структуризации задачи в виде взаимодействующих подзадач в ТК Solver используется аппарат описания подзадач в виде функций различных типов.

Выбор типа функции для представления подзадачи определяется тем, в каком виде существует это решение. Если решение существует в виде алгоритма, то описать такой алгоритм в ТК Solver можно на встроенном бэйсикоподобном языке программирования и оформить это описание в виде ***процедуры-функции***.

В случае, когда решение подзадачи может быть описано декларативным образом в виде множества уравнений и программных отношений, тогда такое решение можно оформить в ТК Solver в виде функций вычислительных моделей (*ВМ-функций*). Если же решение связано с задачей поиска данных, удовлетворяющих тем или иным свойствам, то это можно оформить в виде ***функций-списков*** (табличных, интервальных или интерполяционных).

Организация взаимодействия подзадач в ТК Solver реализуется путем обращения к представляющей подзадачу функции с помощью аппарата параметров и аргументов (формальных и фактических параметров).

В отличие от функций и процедур в других системах программирования, здесь при описании функции множество ее параметров явно (синтаксическим способом – путем записи символа **;**) разбивается на два подмножества: *входных* и *выходных* переменных, т. е. по существу на вход каждой функции поступает *формулировка подзадачи,* задаваемая в виде описания двух последовательностей: известных и неизвестных переменных.

Алгоритм решения подзадачи для функций различных типов будет свой. Для ***процедур-функций*** он уникален и определяется алгоритмом, который описан в теле процедуры-функции. Для ***ВМ-функций*** механизм нахождения решения подзадачи, которую представляет ВМ–функция, един для всех ВМ–функций и определяется он заложенным в ТК Solver ***алгоритмом планирования вычислений***.

Рассмотрим основные правила описания функций пользователя различных типов.

Функции-списки

Функции-списки позволяют задавать двуместные отношения между двумя списками значений, которые представляют ***область определения (domain)*** и ***область значений (range)*** функции-списка.

Области определения и значения функций–списков, тип соответствия между ними для каждой функции-списка задаются при описании вычислительной модели задачи на панели ***List Function Subsheet***. Для вызова этой панели нужно на панели ***Function Sheet*** указать имя функции в поле Name, выбрать тип List в поле Type и щелкнуть правой кнопкой мыши.

Допускается использование следующих типов соответствий в функциях-списках:

- таблицы (тип функции *table*, 1-1 соответствие);
- интервалы (тип функции *step*, 1-N соответствие);

- линейная интерполяция (тип функции *linear*);
- кубическая интерполяция. (тип функции *cubic*).

Список, задающий значения домена функции-списка просматривается, начиная с головы, т.е. с первого элемента и в качестве результата выдается первое встретившееся подходящее решение.

Все типы функций–списков являются одноместными и могут вызываться как процедуры с помощью **CALL** или использоваться в выражениях. Примеры обращения к функциям–спискам:

weight = PTE(element)
call PTE(element;weight)
elasticity = prop2(material)
density = air3(air1(altitude)*air2(temperature))

Для функций–списков обязательным является соблюдение эквивалентности типов данных переменных, используемых при обращении к функции и используемых в ее теле типов значений домена и диапазона. При этом имена списков для доменов и диапазонов, задаваемые в теле функции, могут совпадать с именами аргументов функции при обращении к ней.

Табличные функции-списки (тип функции ***table*** в поле Mapping) описывают бинарные таблицы. Элементами первого столбца (Domain) таблицы являются значения элементов списка, указанного в поле Domain List. Элементы второго столбца (Range) связаны со списком, указанным в поле Range List. Элементы списков могут принимать числовые или символьные значения. Ниже приведен текст табличной функции с именем country_capital, содержащей символьные данные.

========== LIST FUNCTION: country_capital ==========

Comment: Определение столицы страны
Domain List: country
Mapping: table
Range List: capital
Element-- Domain---------- Range----------
1 'Russia 'Moscow
2 'France 'Paris
3 'England 'London

Возможны следующие варианты обращения к функции:

cap = country_capital(ctry)
call country_capital(country;capital)

Интервальные функции-списки (типа ***step***)**.** В функциях данного типа интервалы между соседними элементами из списка-домена рассматриваются как шаги изменения значений элементов из списка-домена, значения элементов из списка-диапазона могут быть произвольными.

При обращении к такой функции значение аргумента должно принадлежать хотя бы одному из интервалов, образуемых парами соседних элементов из списка-диапазона. Если значение аргумента не равно ни одному из элементов интервала, то результатом функции будет значение элемента из списка-диапазона, соответствующего первому элементу интервала, которому принадлежит значение аргумента. Пример интервальной функции:

========== **LIST FUNCTION: xfourth** ==========

Comment: **четвертая степень x**
Domain List: **x**
Mapping: **Step**
Range List: **y**

Element--	Domain----------	Range----------
1	**1**	**1**
2	**2**	**16**
3	**3**	**81**
4	**4**	**256**

Если в модели есть предложение вида: **y = xfourth(x)** и **x=2.5**, то **y** примет значение, равное **16**.

Функции-списки с линейной и кубической интерполяцией. Функции данного типа подобны интервальным функциям. Различие состоит в том, что в качестве результата выдается значение, полученное в качестве линейной (кубической) интерполяции интервала между соседними элементами списка-диапазона и который соответствует интервалу из списка-домена и в который попадает значение аргумента функции при обращении к ней.

Если бы для функции **xfourth** была бы использована линейная интерполяция, (в поле ***Mapping*** указать тип функции ***linear***), то при обращении к ней с тем же значением аргумента **x=2.5** результат будет **48.5**, а если бы кубическая (в поле ***Mapping*** указать тип функции ***cubic*)**, то результат получится равным **38.5**. Из примера функции **xfourth**, а в ней заданы значения для функциональной зависимости **y = x^4**, видно, что целесообразно использовать для данного соответствия кубическую интерполяцию, так как при этом обеспечивается наибольшая точность вычислений (для **2.5^4=39.0625**).

Общие понятия ВМ-функций и процедур-функций

В ВМ-функциях и процедурах-функциях могут использоваться *входные, выходные и промежуточные* переменные. Все переменные в них являются *локальными*, т.е. они не доступны для обработки вне тела процедур и функций. Передача информации в процедуры-функции и ВМ-функции и возврат вычисленных значений из них осуществляется через аппарат *параметров* и *аргументов* (формальных и фактических параметров).

В общем случае все переменные связи при вызове процедуры-функции или ВМ-функции можно разбить на *входные* и *выходные*. При этом ***входные переменные*** в зависимости от способа передачи данных также можно разделить на два вида: *передаваемые при вызове* процедуры-функции или ВМ-функции и *передаваемые непосредственно из* ***Variable Sheet***, минуя список вызова. В описаниях процедур-функций и ВМ-функций все перечисленные виды переменных связи формируются путем соотнесения их в соответствующий список.

При описании ВМ-функций и процедур-функций можно объявить от 0 до 20 переменных каждого вида (входных и выходных). Все объявленные переменные, кроме параметров, являются локальными. При вызове ВМ-функций и процедур-функций соответствие между параметрами и аргументами осуществляется позиционным способом: количество аргументов (входных и выходных) должно соответствовать порядку следования и количеству параметров (входных и выходных) ВМ-функции или процедуры-функции.

Процедуры-функции и ВМ-функции, в описаниях которых объявлен один выходной параметр (одна выходная переменная) (***Output***) или соответственно один результат (***Result***), могут вызываться либо с помощью вызова **CALL**, либо **по ссылке**. В последнем случае ссылка используется в выражениях и количество аргументов при обращении к процедуре-функции или ВМ-функции будет на единицу меньше числа описанных параметров. При вызове по **CALL** в списке аргументов входные переменные отделяются от выходных переменных точкой с запятой.

В процедурах-функциях вход и выход соответственно называют *входными* и *выходными переменными*, в ВМ-функциях их называют *переменными-аргументами* (*Argument Variables)* и *переменными-результатами* (*Result Variables)* соответственно.

ВМ-функции

Каждая ВМ-функция – функция, содержащая описание ВМ (тип функции ***Rule***), должна быть объявлена на панели ***Function Sheet*** и описана на панели ***Rule Function Subsheet.***

Описание ВМ-функции включает в себя множество правил (условных и безусловных предложений вычислимости) и объявление ее ***входных переменных***:

- переменных-параметров ВМ-функции в поле *Parameter Variables* (значения переменным этого поля присваиваются непосредственно из панели *Variables Sheet*);
- переменных-аргументов в поле *Argument Variables* (значения переменных передаются через список вызова ВМ-функции);

и ***выходных переменных*** – результатов функции в поле *Result Variables* (значения возвращаются через список вызова).

В общем случае для ВМ-функций можно отойти от жесткого разделения на только входные или только выходные переменные: входными считаются переменные, значения которых известны до обращения к функции, а выходными – переменные, значения которых могут быть вычислены для заданного множества входных переменных. Поэтому входными могут быть переменные, описанные и как аргументы, и как результаты, аналогично и выходными. Однако в вызове ВМ-функции должен быть специфицирован полностью весь список аргументов и результатов ВМ-функции, даже если заведомо известно, что не все исходные данные означены и соответственно будут получены не все результаты.

В ВМ-функциях попытка означить переменную-параметр значением, отличным от того что задано на панели ***Variables Sheet***, приводит к ошибке, фиксируются ошибки также, если вычисленное значение какой-либо переменной с помощью одного соотношения, не согласуется со значением этой или другой переменной, вычисленной с помощью другого соотношения.

В теле ВМ-функции могут быть ссылки или обращения к другим ВМ-функциям или процедурам (рекурсивные ссылки и вызовы не допускаются, т.е. нельзя обращаться к самой себе).

Решатель задач осуществляет обработку переменных и правил ВМ-функции аналогично тому, как это делается для переменных и правил из панели описания вычислительной задачи ***Rule Sheet***. Неизвестные переменные могут содержаться как в левых, так и в правых частях правил и вычисляются с использованием тех правил, в которых они представлены.

Значения переменных передаются в и возвращаются из ВМ-функций при вызове ВМ-функции по **Call** или с помощью ссылки к ВМ-функции в выражении. Связывание локальных переменных из правил ВМ-функции с переменными, которые указываются при ее вызове по **Call** или по ссылке, осуществляется позиционным способом.

После вызова ВМ-функции в вычислениях участвуют те правила, для которых заданы значения входных переменных через переменные-параметры, переменные-аргументы и переменные-результаты. Ниже приведены примеры описания функций.

ВМ-функции **rat** и **Cone** используют ВМ-функцию **Pythagoras**. ВМ-функция **INTEGRAND** может быть использована в подпрограмме интегрирования (процедуре-функции) для численного вычисления интеграла. При этом **a** является переменной-параметром для ВМ-функции **INTEGRAND** и значение ее передается из ***Variable Sheet.***

========== RULE FUNCTION: Pythagoras ==========
Comment: Теорема Пифагора
Parameter Variables:
Argument Variables: side1,side2
Result Variables: hypotenuse
S Rule----------
hypotenuse^2 = side1^2 + side2^2

========== RULE FUNCTION: rat ==========
Comment: Прямоугольный треугольник
Parameter Variables:
Argument Variables: a,b,c,alpha,beta,perimeter,area
Result Variables:
S Rule----------
alpha + beta = pi()/2
call Pythagoras(a,b;c)
a/b = tan(alpha)
perimeter = a + b + c
area = a * b/2
========== RULE FUNCTION: CONE ==========
Comment: Геометрия конуса
Parameter Variables:
Argument Variables: radius,height,slant,theta
Result Variables: surface,volume
S Rule----------
slant = Pythagoras(radius,height)
tand(theta/2) = radius/height
surface/slant = pi() * radius
volume = radius^2 * height * pi()/3
========== RULE FUNCTION: INTEGRAND ==========
Comment: Интегрируемая функция
Parameter Variables: a
Argument Variables: x
Result Variables: y

S Rule----------

y = a^x – ln(x)

Процедуры-функции

Каждая ***процедура-функция*** (*Procedure Function,* в дальнейшем будем называть ее просто ***процедура***) должна быть объявлена на панели ***Function Sheet*** и определена на панели ***Procedure Function Subsheet***. Описание процедуры включает в свой состав множество операторов и объявления:

- *параметров* (передаются непосредственно из панели ***Variables Sheet***, описываются в поле ***Parameter Variables***);
- *входных переменных* (поле ***Input Variables***) и *выходных переменных* (поле ***Output Variables***), которые передаются через список вызова процедуры.

В процедуре с помощью ***операторов*** бейсикоподобного алгоритмического языка задается алгоритм вычисления значений *выходных переменных*, при этом порядок выполнения операторов процедуры детерминирован. В процедурах допускаются ссылки к другим процедурам и функциям, можно использовать и рекурсию, т.е. обращение по **Call** к самой себе.

Переменные, объявленные как входные (***Input***), должны быть известны до вызова процедуры, хотя значения входных переменных можно изменять внутри процедуры (в BM-функциях это приводит к ошибке вычислений). Однако измене́ия эти не распространяются на соответствующие фактические переменные вызова процедуры, исключение составляют переменные типа список – значения элементов списка могут быть изменены в процедуре, даже если имя списка передается как ***Input***-переменная. Аналогично, в теле процедуры можно изменять значения параметров, но действуют эти изменения также в рамках тела процедуры.

Операторы определяют последовательность вычислений и используются внутри процедур. Рассмотрим правила и примеры записи различных операторов.

Безусловные операторы.

Оператор присваивания имеет вид:

<ИМЯ ПЕРЕМЕННОЙ>:= <ВЫРАЖЕНИЕ>

В качестве имени переменной может использоваться как простая переменная, так и переменная, инициализирующая обращение к элементу списка. В операторе присваивания допускается использовать знак = вместо **:=.**

Оператор безусловного перехода:

GOTO <ИМЯ МЕТКИ>

При выполнении оператора **GOTO** осуществляется передача управления оператору, помеченному меткой:

<ИМЯ МЕТКИ>: <ОПЕРАТОР>

Оператор выхода из процедуры:

RETURN

Оператор вызова процедуры:

CALL <ИМЯ ПРОЦЕДУРЫ> (<ВХОДЫ ПРОЦЕДУРЫ> ; <ВЫХОДЫПРОЦЕДУРЫ>) ·

Оператор цикла:

FOR <ИМЯ ПЕРЕМЕННОЙ ЦИКЛА>:= <ВЫРАЖЕНИЕ1>
TO <ВЫРАЖЕНИЕ2> [STEP <ВЫРАЖЕНИЕ3>]
<ПОСЛЕДОВАТЕЛЬНОСТЬ ОПЕРАТОВ ЦИКЛА>
NEXT [<ИМЯ ПЕРЕМЕНОЙ ЦИКЛА>]

Здесь **<ВЫРАЖЕНИЕ1>** и **<ВЫРАЖЕНИЕ2>** определяют соответственно начальное и конечное числовые значения переменной цикла, а **STEP** задает значение шага изменения переменной цикла, при отсутствии части **STEP** значение шага равно **1**.

Внутри тела цикла могут использоваться вложенные циклы. При организации таких циклов применяются оператор продолжения **CONTINUE** и оператор выхода **EXIT**. *Оператор* **CONTINUE** имеет вид:

CONTINUE [<ИМЯ ПЕРЕМЕННОЙ ЦИКЛА>]

При выполнении данного оператора управление передается оператору **NEXT** текущего цикла либо оператору **NEXT** объемлющего цикла, параметром которого является переменная, указанная в операторе **CONTINUE**.

Оператор выхода **EXIT** задается в виде:

EXIT [<ИМЯ ПЕРЕМЕННОЙ ЦИКЛА>]

и инициирует передачу управления оператору, расположенному за оператором **NEXT** текущего цикла, либо за оператором **NEXT** объемлющего цикла, параметром которого является переменная, указанная в операторе **EXIT**.

Условный оператор задается в следующей форме:

IF <ЛОГИЧЕСКОЕ ВЫРАЖЕНИЕ> THEN <ОПЕРАТОР1>
[ELSE <ОПЕРАТОР2>]

Примеры операторов:

```
1. r = sqrt(x^2+y^2)
2. 'x[i][j] = 0
3. RETURN
4. Loop:
5. GOTO loop_end
6. loop_end: NEXT
7. CALL quicksort('Age,'Height,'Weight;'Fact)
8. = ptor(r1,theta1) + ptor(r2,theta2)
9. IF x>=0 THEN y=sqrt(x)
10. IF log(x)>a THEN y=0 ELSE y='unknown
11. IF x[i][j]=0 THEN return ELSE goto loop
12. IF evltd('x,1,0) THEN y=x^2 ELSE y=z^2
13. FOR i = j TO k
        x[i] = (k-j)*i
        NEXT i
```

Примеры описания и использования процедур:

```
================= PROCEDURE FUNCTION: Simpson =================

Comment:          Определенный интеграл, метод Симпсона
Parameter Variables:
Input Variables:    fun,a,b,n
Output Variables:   value
S Statement----------------------------------------------------------
 ; Обозначения: fun – имя правила, процедуры, списка или встроенной
 ;                  функции определения подынтегрального выражения
 ;             a,b – верхний и нижний пределы
 ;             n – число шагов интегрирования (четное число)
```

```
;              value – значение определенного интеграла
; Описание:  Стандартный метод численного интегрирования
; с полиномиальной аппроксимацией второй степени
;  подынтегрального выражения

if mod(n,2)<>0 then call errmsg("odd numbers of intervals, must be even")
h = (b – a)/n
x = a
k = 1
value = 0
for i = 2 to n
        x = x + h
        k = 3 – k
        value = value + k*apply(fun,x)
next i
value = (2*value + apply(fun,a) + apply(fun,b)) * h/3
```

В приведенном примере **apply** представляет собой встроенную функцию, которая позволяет использовать ее аргумент (**fun**) для ссылки к имени функции, которая будет вычисляться с аргументом **x**. Конкретное значение **fun** определяется при обращении к процедуре-функции **Simpson**.

Пример описания и использования процедуры-функции для решения задачи вычисления 4! приведен на рис. 22.6.

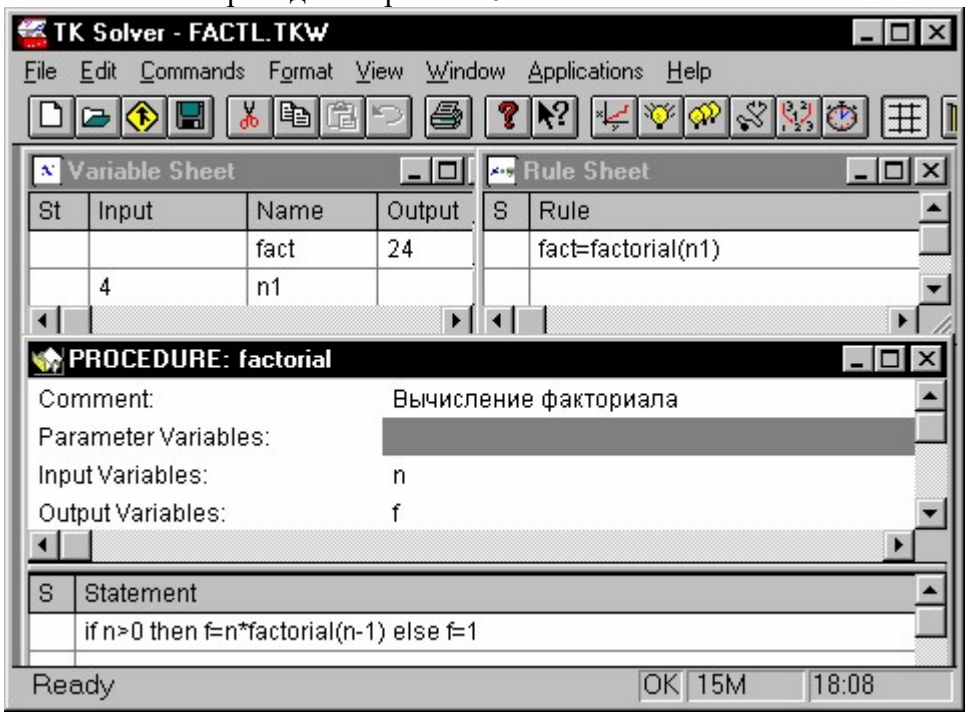


Рис. 22.6. Описание и использование процедуры Factorial

В окне TK Solver (рис. 22.6) после решения задачи показаны три панели: ***Variable Sheet*** с переменными n1 (входная переменная типа Input с значением 4) и fact (выходная переменная типа Output с результирующим значением 24); ***Rule Sheet*** с правилом вычисления значения fact, содержащим обращение к процедуре-функции

Factorial; ***Procedure: Factorial*** с описанием процедуры-функции **Factorial**, предназначенной для вычисления **n!**.

22.5. Списки

Список представляет собой последовательность значений (числовых или символьных), называемых ***элементами*** списка. Для обращения к списку необходимо указать его имя. Доступ к элементам списка осуществляется путем указания имени списка и номера элемента, номер – положительное целое число без знака большее нуля.

Списки в ТК Solver используются различными способами. Они могут ассоциироваться с переменными и использоваться для хранения значений входных данных задачи и получать значения выходных данных задачи в режиме списковых вычислений решателя. Списки служат для хранения аргументов и результатов в списковых функциях. Списки используются для организации связи по данным ТК Solver с другими системами через файлы.

Переменная модели задачи будет иметь тип список, и в модели появляется список с одинаковым с переменной именем, если при объявлении переменной на панели ***Variable Sheet*** в поле ***Status*** указать тип ***L***. После этого каждая ссылка к этому имени во всех компонентах ТК Solver интерпретируется как обращение к списку. В выражениях и аргументах ВМ-функций и процедур-функций к спискам ссылаются путем указания в форме символьного значения, т.е. имени переменной типа список должен предшествовать *апостроф.*

Например:

call blank('abc)

дает тот же результат, что и **call blank(uv)**, если **uv='abc**, здесь **abc –** имя списка.

Чтобы обратиться к элементу списка, необходимо в операторах и предложениях вычислимости использовать конструкцию вида:

<ИМЯ СПИСКА > [<АРИФМЕТИЧЕСКОЕ ВЫРАЖЕНИЕ>]

Например, оператор **'abc[4]:=100** присваивает четвертому элементу списка **abc** значение, равное **100**. Все имена списков текущей модели задачи хранятся в панели ***List Sheet***. Имя нового списка автоматически заносится в панель ***List Sheet***, если:

- переменной на панелях Variable Sheet или Variable Subsheet в поле Status указывается тип L;
- имя списка назначается в поле Associated List на панели Variable Subsheet;
- имя списка назначается в области domain или range на панели List Function Subsheet;
- имя списка назначается на X-Axis, Y-Axis или на панели Plot Subsheet;
- задано имя списка в поле List на панели Table Subsheet.

Новый список появляется в модели задачи также, если переменная в уравнении или операторе принимает значение несуществующего списка. Например, если выполнено правило **place('abc,2)=pi()** или оператор **'abc[2]:=pi(),** то создается список **abc** и второй элемент его принимает значение **3.14159**... . Данное обстоятельство позволяет моделировать многомерные массивы, представимые в этом случае как ***списки списков***. Например, после выполнения оператора присваивания вида **'abc[i][j][k]:= i*j*k**, при **i=4, j=6 и k=5** будет создан главный список **abc**, и его **4**-й элемент будет именовать подсписок с именем **'abc#4**, затем

будет создан список **abc#4**, и в **6**-й его элемент будет помещено имя подсписка нижнего уровня **'abc#4#6**; окончательно будет создан список **abc#4#6** и значение **120 (=i*j*k)** будет присвоено его **5**-му элементу.

22.6. Пары и комплексные числа

Пара представляет собой два арифметических выражения, разделенные запятой и заключенные в круглые скобки:

(<АРИФМЕТИЧЕСКОЕ ВЫРАЖЕНИЕ1>,< АРИФМЕТИЧЕСКОЕ ВЫРАЖЕНИЕ2>)

Пара выражений вида (a,b) используется для представления:

- прямоугольных или полярных координат точки на плоскости;
- комплексных чисел;
- связанных по некоторой причине выражений a и b.

Тип представления определяется контекстом правила или оператора, в котором эта пара используется. Пара выражений может использоваться для получения значений переменных в функциях с двумя выходными параметрами. Например, при выполнении правила **(q,r) = divide(11,3)** вычисленные функцией **DIVIDE()** частное и остаток от деления числа **11** на число **3** присваиваются переменным пары, т.е. будет получено **q=3** и **r=2.**

Примеры использования пар:

1. = (a*cosd(t),b*sind(t))
2. (abserr,relerr) = (a2-a1,(a2-a1)/a1)

Последние два выражения эквиваленты четырем выражениям без использования пар:

1. **x = a*cosd(t)**
2. **y = b*sind(t)**
3. **abserr = a2-a1**
4. **relerr = (a2-a1)/a1**

Пары могут использоваться как в уравнениях при описании вычислительных моделей, так и в операторах процедур-функций. При использовании пар в уравнениях необходимо учитывать возможные конфликты при попытке перевычисления известных значений переменных модели. Пример такой ситуации:

(a,b) = (b,a).

В процедуре-функции данное выражение описывает оператор, который осуществляет переприсваиваивание значений переменных **a** и **b**. В уравнении это недопустимо. Пары также нельзя использовать в качестве операндов логических выражений. Например, следующие условные выражения приводят к ошибкам:

if (a,b) = (1,0) then return
if (u,v) <> (0,0) then z = 1/sqrt(u*v).

Правильная их запись имеет вид:

if and(a=1,b=0) then return
if and(u<>0,v<>0) then z = 1/sqrt(u*v).

В качестве примера использования пары для представления координат точки на плоскости приведем описание параметрической формы задания эллипса:

(x,y) = (a*cosd(t),b*sind(t)),

где **a** и **b** суть константы, задающие полуоси эллипса, **t** – угол в градусах.
Исходное описание **комплексных чисел**, используемое в ТК Solver, имеет вид:

x + iy, где **i^2 = -1**.

Числа **x** и **y** называются ***вещественной*** и ***мнимой*** частями комплексного числа соответственно. Представляются *комплексные числа* в виде пары: **(x,y)**. Над комплексными числами можно выполнять соответствующие арифметические операции: +, -, *, и /. Например: **(a,b) = (c,d)*(e,f)** эквивалентно двум операторам: **a = c*e-d*f и b = c*f+d*e**.
Комплексные числа и операции над ними можно использовать в уравнениях. При этом при решении уравнений для нахождения неизвестных переменных обратные операции будут выполняться также по законам комплексной арифметики. Например, если заданы **a, b, e** и **f**, в предыдущем примере будет выполнено вычисление **(a,b)/(e,f)** для нахождения неизвестных значений **c** и **d**. Смешивать комплексные типы с другими типами в ТК Solver не допускается. Так, запись вида: **(y1,y2) = 2*(z1,z2)** должна быть представлена в форме: **(y1,y2) = (2,0) * (z1,z2).**
Комплексные пары в ТК Solver разрешено использовать в качестве аргументов следующих функциях: **POWER((x,y), n)** возводит комплексное число **(x,y)** в степень **n**, где **n** – вещественное число, необязательно целое; **RE((a,b))** возвращает действительную часть **a** комплексной пары; **IM((c,d))** возвращает мнимую часть **d** комплексной пары.
Точка **z** на комплексной плоскости может быть представлена либо в виде прямоугольных координат **(x,y)**, либо в виде полярных координат **(r,theta)**. Последняя запись в ТК Solver есть представление полярной координаты, а не представление комплексного числа в полярной форме. Полное выражение для точки z как комплексного числа в полярной форме имеет вид **r(cos theta + i sin theta)**.

Контрольные вопросы и задания

1. Укажите отличия парадигм обычных программ и систем искусственного интеллекта.
2. Перечислите модели представления знаний, их достоинтсва и недостатки.
3. Охарактеризуйте типы рассуждений.
4. Сформулируйте правило Modus Ponens и укажите его назначение.
5. В чем суть метода резолюций?
6. Перечислите состав и назначение разделов Пролог-программы.
7. Назовите основные стандартные предикаты Турбо-Пролога.
8. Поясните механизм использования предикатов fail и ! для управления вычислениями.
9. Укажите способы организации рекурсии.
10. Дайте определение списка и поясните основной механизм обработки списков.
11. Укажите состав и структуру системы синтеза программ.
12. Дайте определение и укажите форму задания вычислительной задачи.
13. Охарактеризуйте решатель задач ТК Solver, назовите состав панелей.
14. Назовите разновидности функций и укажите технологию описания функций пользователя.
15. Составить Пролог-программу объединения двух списков.

16.Составить Пролог-программу сортировки элементов списка в порядке убывания.
17.Составить рекурсивную программу вычисления $\Sigma X^{n+1}/(n+1)!$, n=0, 1, 2,..., .
18.Выполнить в системе TK Solver решение следующих систем уравнений:

а) $\begin{cases} X * Y = \sin(X) + 1; \\ Y = \cos(X) + 2. \end{cases}$ б) $\begin{cases} X * Y = \sin(X) + 1; \\ Y - 3 = \cos(X). \end{cases}$

19.Разработать ВМ для расчета параметров пассивных и активных электрических цепей, включающих сопротивления, индуктивности, емкости и источники тока.

Литература

1. Агафонов В. Н., Борщев В. Б., Воронков А. А. Логическое программирование в широком смысле (обзор). В кн. Логическое программирование: Пер. с англ. и фр.- М.:Мир,1988. –368 с.
2. Вольфенгаген В. Э., Яцук В. Я. и др. Методические указания к проведению практических занятий по курсу ”Основы разработки специализированных машин ” (реализация в среде аппликативной логики). – М.: МИФИ,1990. – 60 с.
3. Ин Ц., Соломон Д. Использование Турбо-Пролога: Пер. с англ. – М.: Мир, 1993. – 606 с.
4. Искусственный интеллект: в 3-х кн. Кн. 2. Модели и методы: Справочник/Под ред. Д. А.Поспелова. – М.: Радио и связь, 1990 г. – 304 с.
5. Искусственный интеллект: в 3-х кн. Кн. 3. Программные и аппаратные средства: Справочник/Под ред. В. Н.Захарова, В. Ф.Хорошевского. – М.: Радио и связь, 1990. – 368 с.
6. Искусственный интеллект – основа новой информационной технологии/Поспелов Г. С. – М: Наука, 1988. – 280 с.
7. Компаниец Р. И., Ломако А. Г. Программирование и исчисление вычислительных задач. Инф. бюллетень № 89. – СПб.: ВИККА, 1996. – 29 с.
8. Котенко И. В. Логическое программирование на Прологе. TURBO и PDC PROLOG. Учебное пособие. – СПб.: ВАС, 1993. – 412 с.
9. Лорьер Ж.-Л. Системы искусственного интеллекта: Пер. с франц. – М.: Мир, 1991. – 568 с.
10. Люгер Дж. Ф. Искусственный интеллект: стратегии и методы решения сложных проблем, 4-е издание: Пер. с англ. – М.: Издательский дом "Вильямс", 2003. – 864 с.
11. Марселлус Д. Программирование экспертных систем на Турбо-Прологе.: Пер. с англ. – М.: Финансы и статистика, 1994. – 256 с.
12. Осуга С. Обработка знаний: Пер. с япон. – М.: Мир, 1989. – 293 с.
13. Стерлинг Л., Шапиро Э. Искусство программирования на языке Пролог: Пер.с англ. – М.: Мир,1990. – 235 с.
14. Тыугу Э. Х. Концептуальное программирование. – М.:Наука,1984. – 256 с.
15. Фатиев Н. И. Логика: Учебное пособие. – СПб.: СПбГУП, 1994. – 118 с.