Конспект лекций

С.В. Родионов

Представление знаний в системах искусственного интеллекта

**Лекция 1. Введение. Эволюция систем искусственного интеллекта**

**Ранний этап – использование ЭВМ для решения нечисловых задач**

**Машинный перевод:**

1954 г. "Джорджтаунский эксперимент" – переведено 60 фраз.

* 1-й этап - информация о языках, между которыми выполнялся перевод, и описание самих правил перевода – в едином алгоритме;
* 2-й этап (60-е годы) – вводится специальный язык-посредник, облегчающий сопоставление конструкций языков;
* 3-й этап (2-я половина 70-х) – язык-посредник преобразуется в модель глубинной семантики, описывающей семантические инварианты, присущие всем естественным языкам.

**Автоматизированное реферирование и информационный поиск:**

* подходы, основанные на выявлении статистических закономерностей распределения терминов в тексте или их взаимного расположения в нем;
* использование внутренних структур текста, выявление информационной основы, организующей весь текст.

**Доказательство теорем:**

* середина 50-х г. – с помощью ЭВМ доказаны первые теоремы исчисления высказываний и (чуть позже) исчисления предикатов;
* 1965 г. Дж. Робинсон (США) – метод резолюций;
* 1967 г. – С.Ю.Маслов (СССР) – обратный метод, предназначенный для построения машинных алгоритмов поиска логического вывода;
* Эвристическое программирование.

Эвристическим программированием названы такие методы поиска оптимальных решений, основу которых составляют формализованные (т.е. представленные в виде конечного алгоритма) эвристики, причем под эвристикой понимаются методы отыскания нового.

Под эвристикой принято понимать правило, стратегию или просто ловкий прием, найденные человеком на основе своего опыта, имеющихся знаний и интуиции и позволяющие наиболее эффективно решать некоторый класс слабоструктурированных задач

**Распознавание образов:**

* обучение нахождению решающего правила на множестве положительных и отрицательных примеров;
* логико-лингвистические методы распознавания.

**Игровые программы:**

* конец 40-х годов – программы для простых игр типа "крестики-нолики", "ханойская башня";
* Программы для игры в домино, шашки, шахматы, карточные игры и др. – *нахождение эффективных стратегий поиска по дереву игры*

**Сочинение музыки и текстов.**

**1956 г. - появление ИИ как научного направления** (конференция в Дартмуте (США)

**Начальный этап развития:**

* Лабиринтная модель решения задач.
* Ситуационное управление*.*
* Не существует какого-либо универсального подхода к управлению. Разные проблемные ситуации требуют различных подходов к их разрешению.
* Ситуационные вероятностные факторы учитываются в стратегиях, структурах и процессах, благодаря чему достигается эффективное принятие решений.
* Существует более одного пути достижения цели.
* Результаты одних и тех же управленческих решений могут существенно отличаться друг от друга.
* Всякая управленческая проблема должна рассматриваться только в тесной связи с другими проблемами.
* Менеджеры могут приспосабливать свои организации к ситуации или изменять ситуацию согласно требованию организации.
* Управление — это прежде всего искусство менеджера правильно определить и оценить ситуацию и выбрать наиболее эффективные методы управления, наилучшим образом отвечающие возникшей ситуации.

Ассоциативная модель.

* Это предложенная Саймоном Уильямсом модель представления данных, в которой база данных состоит из двух типов структур данных — элементов и ссылок, хранимых в единой однородной общей структуре в качестве альтернативы [реляционной](http://dic.academic.ru/dic.nsf/ruwiki/41346) и объектной моделям данных. Близка к модели данных [cущность-связь](http://dic.academic.ru/dic.nsf/ruwiki/147094).

## Существо модели

* Реальный мир моделируется компонентами двух сортов: *сущностями* (элементами) и *ассоциациями* (связями). Сущность - это нечто, существующее отдельно и самостоятельно. Ассоциация - это нечто, существующее только в связи с какими-то другими вещами, и которое не имеет смысла в отрыве от них
* Нейробионический подход

В его основе лежат системы элементов, способные подобно нейронам головного мозга воспроизводить некоторые интеллектуальные функции. Прикладные системы, разработанные на основе этого подхода, называются нейронными сетями

**Создание теоретической базы**

*Представление знаний.* Основные задачи**:**

* методы сбора знаний о проблемной области
* методы представления знаний в базе знаний в форме, удобной для обработки на ЭВМ;
* методы сохранения непротиворечивости и достижения полноты знаний при объединении знаний, получаемых из различных источников;
* методы классификации собранных знаний и их обобщения в процессе накопления;
* методы использования знаний при решении различных задач.

*Общение.* Методы организации интеллектуального человеко-машинного взаимодействия.

*Рассуждения и планирование.*

* принятие решений в альтернативных ситуациях;
* нахождение и обоснование планов целесообразной деятельности.

*Восприятие.* Задачи обработки образов, требующие использования знаний.

*Обучение.* Использование информации о подтверждении или неподтверждении некоторых гипотез фактами, хранящимися в базе знаний ИС.

**Переход к промышленным образцам**

Начало 80-х годов характеризуется изменением взгляда на ИИ. На смену "игрушечным" моделям и чисто демонстрационным интеллектуальным програм­мам стали приходить системы, интересные и важные для решения трудных практи­ческих задач, которые не могли быть решены ранее известными методами.

Появилось семейство языков программирования, не ориентированных на решение чисто вычислительных задач (ЛИСП и ПРОЛОГ).

Стали активно развиваться различные *инстру­ментальные системы поддержки разработок интеллектуальных систем,* даю­щие программистам возможность автоматизировать свою деятельность при создании новых интеллектуальных программ и систем.

**Области практического приложения интеллекту­альных систем, созданных в первой половине 80-х годов:**

* Традиционные системы управления.
* роботизированные производства с гибкой технологией,
* распределенные интеллектуальные системы (пример - отраслевое планирование).

– Автоматизация научной и инженерной деятельности.

* Расчетно-логические системы с развитой графической системой общения,
* Обучающие системы (тьюторы),
* Экспертные системы,
* Производство ЭВМ новых поколений (ЭВМ 5-го поколения должны быть такими, чтобы пользователь мог их применять с такой же легкостью, с какой он пользуется другими приборами, носящими название бытовых).

К началу 90-х годов на центральное место в исследованиях по ИИ выдвинулась концепция *интеллектуального агента* (ИА).

Определения ИА:

* любая *сущность*, *воспринимающая* свою *среду* посредством сенсоров и *воздействующая на нее* посредством эффекторов [Russell and Norvig, 1995].
* *система*, *помещенная в* некоторую *среду* и *являющаяся ее частью*, постоянно *воспринимающая* эту *среду*, *воздействующая на нее* для решения своих задач и *определяющая*, *что* ей *воспринимать в будущем* [Franklin and Graesser, 1996].
* системы, непрерывно реализующие *три функции* [Hayes-Roth, 1995]:
  + *восприятие* динамически меняющегося состояния внешней среды;
  + *рассуждения* для интерпретации воспринимаемой информации, решениязадач, реализации вывода и определениядействий;
  + *воздействие на среду* с целью изменения ее состояния
* программные сущности, выполняющие некоторое множество операций в интересах пользователя или других программ с определенной степенью независимости или автономности и использующие для этого знания или представления о целях и желаниях пользователя.
* вычислительная система, обитающая в некоторой сложной динамической среде, воспринимающая ее, и действующая автономно, чтобы реализовать множество целей или задач, для которых она создана [Maes, 1995].

Общее: *активное взаимодействие со средой* и *целенаправленное поведение*.

Наиболее распространенное определение ИА [Wooldridge and Jennings, 1995]:

Система, обладающая следующим набором свойств:

* *автономность* – способность действовать (решать задачи) без прямого вмешательства человека или других агентов, самостоятельно управляя своими действиями и внутренним состоянием;
* *реактивность* – способность к активному восприятию внешней среды и своевременному отклику на происходящие в ней события (внешняя среда агента может быть физическим миром, пользователем, совокупностью других агентов или их комбинацией);
* *активность* – способность осуществлять целенаправленное поведение и проявлять инициативу, а не просто откликаться на внешние события;
* *социальность* – способность взаимодействовать с другими агентами и/или людьми, общаясь с ними посредством языка(ов) межагентного общения.

Это *слабое* определение, т.к. оно характеризует внешние свойства агента и ничего не говорит о его внутренней организации.

*Сильное определение* рассматривает агента как *«интенсиональную систему»*, т. е. систему, при описании внутреннего состояния и процесса переработки информации которой используются так называемых *«ментальные категории»*. К таким категориям принято относить: *знания* (*knowledge*), *убеждения* (*beliefs*), *желания* (*desires*), *цели* (*goals*), *намерения* (*intentions*), *планы* (*plans*), *обязательства* (*commitments*) и т. п.

Термин *«интенсиональная система»* (*intentional system*), широко используемый в теории агентов,был введен философом Д. Деннетом [Dennett, 1987] для описания сущностей «поведение которых может быть предсказано путем приписывания им убеждений, желаний и целесообразности».

Существенной особенностью ИА является способность осуществлять *целенаправленное поведение* в *открытых, динамических и неопределенных мирах*.

*Открытые* миры, в отличие от замкнутых, не позволяют на этапе проектирования агента исчерпывающе полно описать среду его функционирования, т.к. неизвестно, сколько, каких сущностей агент обнаружит в окружающей среде. Агент должен обладать общими знаниями, позволяющими ему распознавать различные типы сущностей, их поведение, намерения и динамически строить планы собственных действий.

*Динамичность* мира означает, что его состояние постоянно меняется. Источниками изменений мира быть как целенаправленные сущности (другие агенты), так и нецеленаправленные динамические объекты.

*Неопределенность* – означает принципиальную невозможность для агента исчерпывающе полно и точно идентифицировать состояния мира. Существуют различные типы неопределенности знаний. При функционировании в мультиагентных средах важную роль играет неопределенность относительно целей других агентов, что обуславливает ограниченную предсказуемость развития событий. Следует также учитывать, что процесс сбора информации требует определенного времени, тогда как в динамической среде ситуация непрерывно меняется. Поэтому агенту необходимо уметь постоянно фокусировать свое внимание на наиболее значимых для него объектах и событиях.

Таким образом, ИА можно также определить как основанную на знаниях систему, способную реализовывать автономное целенаправленное поведение в открытых, динамических и неопределенных мирах, в том числе населенных другими целенаправленными агентами.

**Лекция 2. Программная среда разработки ЭС CLIPS: Назначение и основные возможности. Базовые типы данных и представление фактов.**

**Общая характеристика среды CLIPS**

Среда CLIPS (C Language Integrated Production System) предназначена для построения ЭС и поддерживает три основных способа представления знаний:

* продукционные правила для представления эвристических знаний;
* функции для представления процедурных знаний;
* объектно-ориентированное программирование.

Поддерживаются 6 основных черт ООП: классы, обработчики сообщений, абстракции, инкапсуляция, наследование и полиморфизм. Правила могут сопоставляться с объектами и фактами.Приложения могут разрабатываться с использованием только правил, только объектов или их комбинации. Предусмотрена интеграция с другими средствами: CLIPS может вызываться из процедурных языков, выполнять свои функции и затем возвращать управление вызвавшей программе. С другой стороны, процедурный код может быть определен как внешняя функция и вызван из CLIPS.

Первоначально CLIPS поддерживал возможность для представления только правил и фактов. Начиная с версии 6.0, правила могут сопоставляться с объектами, так же как с фактами. Более того, объекты могут использоваться безправил путем посылки сообщений и в этом случае отпадает необходимость в машине вывода, если используются только объекты.

**Базовые типы данных и представление фактов в системеCLIPS**

В CLIPS поддерживаются восемь базовых типов данных: *целые* (integer) и *вещественные* (float) числа, *символьные* (symbol) и *строковые* (string) данные, *внешний адрес* (external-address), *адрес факта* (fact-address), *имя экземпляра* (instance-name) и *адрес экземпляра* (instance-address).

*Целые числа* состоят из знака (необязательного для положительных чисел) и последовательности десятичных цифр. Например: 27; +125; -38.

*Вещественные* числа содержат мантиссу (в которой дробная часть отделяется точкой) и необязательный порядок, состоящий из символа "e" и целого числа. Например: 12.0; -1.59; 237e3; -32.3e-7.

*Символьное* значение задается продолжающейся до ограничителя последовательностью отображаемых ASCII-символов. Ограничителями являются: любой неотображаемый ASCII-символ (пробел, табуляция, возврат каретки, перевод строки), кавычка, открывающая и закрывающая скобки, амперсанд (&), вертикальная черта (⏐), знак «меньше» (<) и тильда (~). Точка с запятой (;) также является ограничителем и используется для указания на начало комментария. Ограничители не включаются в символьное значение, за исключением символа "<", стоящего в начале значения. Символьное значение не может начинаться с символа "?" или пары символов "$?", но может содержать их внутри себя. CLIPS различает символы нижнего и верхнего регистров, поэтому abc и ABC воспринимаются как два разных значения. Примеры символьных значений: bad\_value; 456-93-039; @+=%.

*Строковое* значение – это заключенная в кавычки последовательность (в том числе пустая) отображаемых символов. Чтобы включить кавычки в строковое значение, перед ними необходимо поставить символ "\". Чтобы вставить в строковое значение символ "\", перед ним необходимо поместить еще один символ "\", т.е. записать два символа "\" подряд. Примеры строковых значений: "abc"; "a & b"; "a\"quote"; "fgs\\85".

*Внешний адрес* – адрес внешней структуры данных, возвращаемый интегрированной в CLIPS функцией (написанной на языках C или Ada). Этот тип данных создается только как результат вызова функции, его невозможно специфицировать вводом значения. Отображаемое представление внешнего адреса:

<Pointer-XXXXXX>,

где ХХХХХХ – внешний адрес.

*Адрес факта* используется для ссылки на факты. Факт представляет собой список атомарных значений, на которые можно ссылаться либо позиционно (в упорядоченных фактах), либо по имени (в неупорядоченных фактах). Отображаемый формат адреса факта:

<Fact-XXX>,

где ХХХ – индекс факта.

*Имя экземпляра* используется для ссылки на экземпляры классов. *Экземпляр* представляет собой *объект*, являющийся представителем некоторого *класса*. Объектами в CLIPS по определению являются целые и вещественные числа, символьные и строковые значения, многоместные значения, внешние адреса, адреса фактов или экземпляры определенных пользователем классов (создаваемых с помощью конструкции defclass). *Имя экземпляра* представляется заключенным в квадратные скобки символьным типом. Например: [pump-1]; [foo]; [+++]; [123-890]. Скобки не являются частью имени, а только указывают тип значения.

*Адрес экземпляра* может быть получен путем связывания значения, возвращаемого функцией instance-address, или связывания переменной с экземпляром, сопоставляемым с объектным образцом в левой части правила Невозможно специфицировать адрес экземпляра вводом значения. Отображаемое представление адресаэкземпляра в CLIPS:

<Instance-XXX>,

где ХХХ – имя экземпляра.

На экземпляры определяемых пользователем классов можно ссылаться либо по имени либо по адресу. Адреса экземпляров должны использоваться, когда критично время решения.

В CLIPS место, занимаемое одним значением базового типа данных, называется *полем* (field). Все значения базовых типов являются *одноместными* (single-field value). Последовательность из нуля или более одноместных значений рассматривается как *многоместное значение* (multifield value). Они отображаются в скобках, причем одноместные значения разделяются пробелами. Примеры многоместных значений: (a 123); (); (x 3.0 "red" 567).

Одной из основных форм представления информации в CLIPS-системах являются *факты*. Они используются правилами для вывода новых фактов из имеющихся. Все текущие факты в CLIPS помещаются в список фактов (fact-list). По формату представления в CLIPS выделяют два типа фактов: *упорядоченные* и *неупорядоченные*.

*Упорядоченный факт* состоит из заключенной в скобки последовательности одного или более разделенных пробелами полей, причем первое поле должно быть символьного типа, а остальные могут быть любыми базовыми типами данных. Первое поле специфицирует отношение, которое применяется к остальным полям факта. Примеры упорядоченных фактов: (высота 100); (студент Сидоров); (отец Иван Петр); (однокурсники Иванов Петров Сидоров).

Для работы с фактами используются следующие команды: assert – добавляет факт в факт-список; retract – удаляет факт из списка; modify – модифицирует список; duplicate – дублирует факт. Например, команда

(assert (length 150) (width 15) (weight “big”))

добавляет в список фактов три факта, каждый из которых состоит из двух полей.

Эти команды могут исполняться как в режиме командной строки, так и включаться в CLIPS-программы. Команды retract, modify и duplicate требуют, чтобы факты были идентифицированы с помощью индекса факта (fact-index) либо адреса факта (fact-address).

*Индекс факта* представляет собой уникальный целочисленный индекс, приписываемый факту всякий раз, когда факт добавляется (или модифицируется). Индексация фактов начинается с нуля и инкрементируется при каждом новом или измененном факте.

*Идентификатор факта* (fact-identifier) представляет собой краткую нотацию для отображения факта. Он состоит из символа “f”, за которым через тире следует индекс факта. Например, f-10 ссылается на факт с индексом 10.

Для задания исходного множества фактов используется конструкция deffacts, имеющая следующий синтаксис:

(deffacts <имя\_группы\_фактов> ["<комментарий>"] <факт>\*),

где <имя\_группы\_фактов> – идентификатор символьного типа;

<комментарий> – необязательное поле комментария;

<факт>\* – произвольная последовательность фактов, записанных через разделитель.

Пример использования конструкции deffacts:

(deffacts stud "Студент"

(student name John)

(student spec “COMPUTER”)).

Факты, определенные конструкцией deffacts, добавляются в список фактов всякий раз при выполнении команды reset.

*Неупорядоченные факты* представляют собой список взаимосвязанных именованных полей, называемых *слотами*. Наличие имен полей позволяет выполнять доступ к полям по именам, в отличие от упорядоченных фактов, где поля специфицируются своим местоположением в факте. Существует два типа слотов: одиночные и мультислоты. Одиночный слот (или просто слот) содержит единственное поле, тогда как мультислот может содержать любое число полей.

Для спецификации состава неупорядоченных фактов (т.е. содержащихся в них слотов) используются *шаблоны*, которые задаются конструкцией deftemplate. Синтаксис конструкции deftemplate определен ниже:

(deftemplate <имя шаблона> [“<комментарий>”]

<определение слота-1>

. . .

<определение слота-N>)

Пример шаблона, содержащего три одиночных слота, представлен ниже:

(deftemplate object “Шаблон объекта”

(slot name)

(slot location)

(slot weight))

Пример конкретного неупорядоченного факта на основе данного шаблона представлен ниже:

(object (name car) (location 100) (weight 600)).

**Лекция 3. Представление и обработка продукций в CLIPS.**

**Представление правил в базе знаний. Типы условных элементов.**

Правила являются основным способом представления знаний в CLIPS. Для задания правил используется конструкция defrule со следующим синтаксисом:

(defrule <имя\_правила> ["<комментарий>"]

[<объявление>]

<условный элемент>\*;Левая часть правила (антецедент)

=>

<действие>\*) ;Правая часть правила (консеквент)

где <имя\_правила> - идентификатор символьного типа, уникальный для данной группы правил; <комментарий> - необязательное поле комментария; <объявление> - необязательный элемент, позволяющий задавать дополнительные свойства правила (например, значимость) с помощью оператора declare; <условный элемент>\* - произвольная последовательность условных элементов; <действие>\* - произвольная последовательность действий.

Пример задания правила:

(defrule R1

(days 2)

(works 100)

=>

(printout t crlf "Свободного времени нет" crlf)

(assert (freetime "no"))).

В левой части правило содержит два условных элемента, сопоставляемых с упорядоченными фактами, а в правой – две команды. В команде вывода сообщений printout параметр t задает стандартный режим вывода, а crlf – символ возврата и перевода курсора на новую строку. Команда assert добавляет в факт-список новый упорядоченный факт.

Антецедент правила состоит из последовательности условных элементов (УЭ). Если все УЭ правила удовлетворяются при текущем состоянии базы данных, то правило помещается в список готовых к выполнению правил –агенду. В CLIPS используется **шесть** типов условных элементов:

* УЭ-образцы (Pattern Conditional Elements);
* УЭ-проверки (Test Conditional Elements);
* УЭ “ИЛИ” (Or Conditional Elements);
* УЭ “И” (And Conditional Elements);
* УЭ “НЕ” (Not Conditional Elements);
* УЭ “Существует” (Exists Conditional Elements);
* УЭ “Для всех” (Forall Conditional Elements);
* логические УЭ (Logical Conditional Elements).

*УЭ-образец*состоит из совокупности ограничений на поля, масок (wildcards) полей и переменных, используемых при сопоставлении УЭ с образцом – фактом или экземпляром объекта. УЭ-образец удовлетворяется каждой сущностью, которая удовлетворяет его ограничениям. *Ограничения на поля* используются для проверки одного поля или слота факта либо экземпляра объекта. Ограничение на поле может состоять из единственного литерала или из нескольких связанных ограничений. В УЭ-образцах используются следующие конструкции:

* литеральные ограничения (Literal Constraints);
* одно и многоместные маски (Single- and Multifield Wildcards);
* одно и многоместные переменные (Single- and Multifield Variables);
* ограничения со связками (Connective Constraints);
* предикатные ограничения (Predicate Constraints);
* ограничения возвращаемым значением (Return Value Constraints).

*Литеральное ограничение* не содержит переменных и масок, а задает точное значение (константу целого, вещественного, символьного или строкового типа, либо имя экземпляра), которое должно сопоставляться с полем образца. Все литеральные ограничения должны совпадать с соответствующими полями сопоставляемой сущности. Упорядоченный УЭ-образец содержит только литералы и имеет следующий синтаксис: (<constant-1> ... <constant-n>). Например, (data 1 one “two”). Пример УЭ-образца для неупорядоченных фактов: (person (name Bob) (age 20)).

*Одно- и многоместные маски*позволяют игнорировать некоторые поля в процессе сопоставления. *Одноместная маска* обозначается символом “?” и сопоставляется с любым значением, занимающим точно одно поле в соответствующем месте сопоставляемой сущности. *Многоместная маска*, обозначается парой символов “$?” и сопоставляется с любыми значениями, занимающими произвольное число полей в сопоставляемой сущности. Маски могут использоваться в одном образце в любых комбинациях. Не допускается лишь использование многоместной маски в одноместном слоте (содержащем единственное поле) неупорядоченных фактов или объектов. Например,УЭ (data ? blue red $?) будет сопоставляться со следующими упорядоченными фактами:

(data 1 blue red),

(data 5 blue red 6.9 "avto"),

но не будет сопоставлен со следующими фактами:

(data 1.0 blue "red"),

(data 1 blue).

*Одно- и многоместные переменные*используются для запоминания значений полей, с целью их дальнейшего использования в других условных элементах антецедента или в операторах консеквента правила. Одноместные переменные начинаются с символа “?”, за которым следует символьное значение, начинающееся с буквы. Например: ?x, ?var, ?age. Многоместные переменные начинаются с префикса “$?”, за которым также следует символьное значение, начинающееся с буквы. Например: $?y, $?zum. В именах переменных не допускается использование кавычек. При первом появлении переменная работает так же, как в маске, т.е. связывается с любым значением в данном поле(ях). Последующие появления переменной требуют, чтобы поле сопоставлялось со связанным значением переменной. Имена переменных являются локальными в пределах каждого правила.

Пусть имеется три факта:

(data 2 blue green),

(data 1 blue),

(data 1 blue red)

и правило:

(defrule find-data-1

(data ?x ?y ?z)

=>

(printout t ?x " : " ?y " : " ?z crlf))

УЭ данного правила будет сопоставляться с первым и третьим фактом, поэтому в результате срабатывания правила будет выведено:

1 : blue : red

2 : blue : green

*Ограничения со связками* используются для связывания индивидуальных ограничений и переменных друг с другом с помощью связок & (“и”), | (“или”) и ~ (“не”), используемых в традиционном смысле. Старшинство операций обычное, за исключением случая, когда первым ограничением является переменная, за которой следует связка &. В этом случае первая переменная трактуется как отдельное ограничение, которое также должно удовлетворяться. Например, ограничение ?x&red⏐blue трактуется как ?x&(red⏐blue), а не как (?x&red)⏐blue.

Пример правила с УЭ, содержащим ограничения со связками:

(defrule r1

(data (value ?x&~red&~green))

=>

(printout t "slot value = " ?x crlf)).

Например, для факта (data (value blue)) это правило выведет сообщение:

slot value = blue.

*Предикатное ограничение* позволяет ограничить значение поля, основываясь на истинности булевого выражения. Для этого используется предикатная функция, которая вызывается в процессе сопоставления с образцом и возвращает в случае неудачи символьное значение FALSE. Если возвращается значение FALSE, то ограничение не удовлетворяется, в противном случае оно удовлетворяется. Предикатное ограничение задается с помощью символа “:”, за которым следует вызов предикатной функции. Данное ограничение может использоваться в комбинации с ограничением со связками, а также связанной переменной. В последнем случае переменная сначала связывается некоторым значением, а затем к ней применяется предикатное ограничение. В таком варианте предикатные ограничения часто применяются для проверки типов данных. При этом в качестве предикатных функций используются встроенные функции CLIPS, в частности:

(numberp <выражение>) – функция возвращает значение TRUE, если <выражение> имеет числовой тип (integer или float), в противном случае возвращается символ FALSE;

(floatp <выражение>) – функция возвращает значение TRUE, если <выражение> имеет тип float, иначе возвращается символ FALSE;

(integerp <выражение>) – функция возвращает значение TRUE, если <выражение> имеет тип integer, иначе – символ FALSE;

(symbolp <выражение>) – функция возвращает значение TRUE, если <выражение> имеет тип symbol, иначе – символ FALSE;

(stringp <выражение>) – функция возвращает значение TRUE, если <выражение> имеет тип string, иначе – символ FALSE.

Пусть заданы факты: ((data 1) (data 2) (data red)). Тогда для определения значений числового типа можно использовать следующий УЭ

(data ?x&:(numberp ?x)),

который сопоставится с первыми двумя фактами. Тот же результат может быть получен использованием УЭ (data ?x&~:(symbolp ?x)).

*Ограничение возвращаемым значением* использует в качестве ограничения значение, возвращаемое внешней функцией. Эта функция вызывается непосредственно из УЭ-образца с использованием следующего синтаксиса:

=<вызов-функции>

Возвращаемое функцией значение одного из базовых типов подставляется непосредственно в УЭ-образец на позицию, из которой была вызвана функция, и используется далее как литеральное ограничение. Например, следующее правило, содержащее УЭ-образец с ограничением возвращаемым значением**:**

(defrule twice

(data (x ?x) (y =(\* 2 ?x)))

=>...)

будет сопоставляться со всеми неупорядоченными фактами, у которых значение в слоте y равно удвоенному значению слота x.

*Условный элемент-проверка* имеет следующий синтаксис:

(test <function-call>).

УЭ-проверка удовлетворяется, если функция, вызываемая из него, возвращает значение, отличное от FALSE. Как и в предикатном ограничении, можно сравнивать уже связанную некоторым значением переменную, используя любые функции (алгебраическое и логическое сравнение, вызов внешних функций). В УЭ-проверку могут быть встроены внешние функции любого вида. В следующем правиле проверяется, что модуль разности двух чисел не меньше трех:

(defrule example-1

(data ?x)

(value ?y)

(test (>= (abs (- ?y ?x)) 3))

=>...)

*Условный элемент “ИЛИ”* задается следующей конструкцией:

(or <УЭ-1> ... < УЭ-N>)

и удовлетворяется, если удовлетворяется хотя бы один УЭ внутри этой конструкции. Наличие такого УЭ позволяет сократить число правил, т.к. тоже самое можно было бы записать множеством правил с одинаковой правой частью. При этом правило будет активизироваться несколько раз,по числу удовлетворяемых комбинаций. Например, правило

(defrule r1

(man stud)

(or (spec computeer) (age 20))

=>...)

эквивалентно двум следующим:

(defrule r2

(man stud)

(spec computeer)

=>...)

(defrule r3

(man stud)

(age 20)

=>...)

*Условный элемент “И”* задается следующей конструкцией:

(and <УЭ-1> ... < УЭ-N>)

и удовлетворяется, если удовлетворяются все УЭ внутри этой конструкции. В CLIPS все УЭ в антецедентах правил неявно объединены по “И”, однако использование УЭ “И” для явного задания конъюнктивной связи позволяет комбинировать УЭ “И” и УЭ “ИЛИ” в любых сочетаниях. Пример такой комбинации приведен в следующем правиле:

(defrule r1

(sys-mode search)

(or (and (distance high) (resol little))

(and (distance low) (resol big)))

=>...)

*Условный элемент “НЕ”* задается следующей конструкцией:

(not <УЭ>)

и удовлетворяется, если содержащийся внутри него УЭ не удовлетворяется. Предварительно связанные переменные могут использоваться внутри УЭ “НЕ” как свободные. Однако, переменные, которые связываются внутри УЭ-“НЕ”, могут использоваться только в этом образце. Следующее правило ищет факты, у которых второе поле – red, а третье и четвертое поля не совпадают:

(defrule not-double

(not (data red ?x ?x))

=>...)

*Условный элемент “Существует”* имеет следующий синтаксис:

(exists <УЭ-1> ... <УЭ-N>)

и используется для определения, удовлетворяется ли группа УЭ, специфицированных внутри условного элемента “Существует”, хотя бы одним набором образцов-сущностей в базе данных. Например, правило:

(defrule example

(exists (a ?x) (b ?x))

=>...)

будет активизировано, если в базе данных имеется хотя бы одна пара фактов, содержащих в первых полях значения a и b, а вторые поля которых совпадают.

*Условный элемент “Для всех”*имеет следующий синтаксис:

(forall <УЭ-1> ... <УЭ-N>)

и используется для определения, удовлетворяется ли группа УЭ, специфицированных внутри условного элемента “Для всех”, для каждого появления УЭ-1. Например, следующее правило активизируется, если каждый студент научился чтению, письму и арифметике:

(defrule all-students-passed

(forall (student ?name)

(reading ?name)

(writing ?name)

(arithmetic ?name))

=>

(printout t "All students passed." crlf))

*Логические условные элементы* обеспечивают возможность *поддержания истинности* различных сущностей (фактов и экземпляров), создаваемых правилами, использующими логические УЭ. Сущность-образец, создаваемая оператором правой части правила, может быть сделана логически зависимой от сущностей-образцов, сопоставляемых с *логическим* УЭ в антецеденте правил. Сущности-образцы, сопоставляемые с логическими УЭ в антецеденте правил обеспечивают *логическую поддержку* фактам и экземплярам, создаваемым в консеквенте правила. Сущность-образец может логически поддерживаться несколькими группами сущностей-образцов из одного или различных правил. Если любая поддерживающая сущность удаляется из группы поддерживающих сущностей и не существует никаких других поддерживающих групп, то поддерживаемая сущность удаляется из рабочей памяти.

Сущность-образец имеет безусловную поддержку, если она создается без логической поддержки, т.е. с помощью конструкций deffacts, definstaces, с помощью высокоуровневых команд или правил без логической поддержки образцов. Безусловная поддержка сущности удаляет всю логическую поддержку (без удаления самой сущности), при этом дальнейшая логическая поддержка безусловно поддерживаемой сущности игнорируется.

Удаление правила, генерировавшего для сущности логическую поддержку, удаляет логическую поддержку, генерируемую этим правилом, но не влечет удаления сущности, даже если для нее не осталось логической поддержки. Логический УЭ имеет следующий синтаксис:

(logical <УЭ>+)

Логический УЭ группирует образцы точно так же, как УЭ “И” и может использоваться в сочетании с УЭ “И”, УЭ “ИЛИ” и УЭ “НЕ”. Однако логические УЭ можно применять только в первых образцах правила. Например, следующее правило допустимо:

(defrule ok

(logical (a))

(logical (b))

(c)

=>

(assert (d)))

Вместе с тем, следующее правило является недопустимым:

(defrule not-ok-1

(logical (a))

(b)

(logical (c))

=>

(assert (d)))

**Лекция 4. Понятие экспертной системы. Основные особенности, архитектура и классификация ЭС. Этапы разработки и стадии жизненного цикла ЭС.**

Понятие экспертной системы.

*Экспертная система* (ЭС) – компьютерная система, использующая знания эксперта для высокоэффективного решения задач в проблемной области, для которой традиционные формальные методы решения неизвестны или неприменимы вследствие имеющихся ограничений.

Первые ЭС начали разрабатываться в середине 60-х годов прошлого века для решения задач медицинской диагностики (система MYCIN), определения структуры сложных молекул по данным масс-спектрограмм (система DENDRAL), определения залежей полезных ископаемых (система PROSPECTOR) и др. В течение 70-х и 80-х годов прошлого века шло активное развитие и формирование инженерии знаний, как важнейшего направления в рамках искусственного интеллекта (ИИ). В настоящее время ЭС широко используется в самых различных областях.

Отличительной чертой данного класса систем является использование для решения задач знаний опытного эксперта.

Кассы задач, в которых используются ЭС:

* *интерпретация* – составление смыслового описания ситуации по наблюдаемым данным – распознавание образов, понимание речи и т. п. (SPE - определение концентрации гамма-глобулина в крови);
* медицинская и техническая *диагностика* – определение причин неисправностей по результатам наблюдений (MYCIN - диагностика бактериальных инфекций);
* *прогнозирование* – определение вероятных последствий наблюдаемых ситуаций – предсказание погоды, урожая, курса валют и т.п. (PLANT/cd - определения потерь урожая от черной совки);
* *планирование* – определение последовательности действий, приводящих к желаемой цели – планирование действий робота, маршрута движения (TATR - планирование авиаударов по аэродромам противника);
* *управление* – целенаправленное воздействие на объект (применяется в задачах, где традиционные модели автоматического управления неприменимы или неэффективны: управление деловой активностью, боем, воздушным движением и т.п.);
* *мониторинг* – сравнение результатов наблюдений с ожидаемыми или желаемыми (медицинский и экологический мониторинг, атомные электростанции);
* *обучение* – диагностика, формирование и коррекция знания и навыков обучаемого GUIDON - обучение студентов-медиков (антибактериальная терапия);.
* о*тладка* - составление рецептов исправления неправильного функционирования системы. ONCOCIN - планирование химиотерапевтического лечения;
* р*емонт* - выполнение последовательности предписанных исправлений. TQMSTUNE - настройка масс-спектрометра.
* *проектирование* - построение конфигурации объектов при заданных ограничениях. XCON (R1) - выбор оптимальной конфигурации аппаратных средств (VAX).

На рис. 13.1 представлена обобщенная архитектура ЭС.



*Интерфейс ЭС с внешней средой* поддерживает взаимодействие ЭС с внешним миром на всех стадиях жизненного цикла системы и включает две компоненты: *интерфейс разработчика* и *интерфейс пользователя*. Интерфейс разработчика используется на этапе разработки ЭС, его основной функцией является поддержка процесса наполнения базы знаний (БЗ). Обычно эта функция выполняется экспертом в предметной области во взаимодействии с инженером по знаниям. Интерфейс пользователя поддерживает общение пользователя с системой в режиме консультации или взаимодействие ЭС с техническими средствами (в случае ее встроенного применения) на этапе ее использования.

*Компонента приобретения знаний* предназначена для занесения в БЗ новых знаний и модификации имеющихся, как на этапе начального обучения ЭС, так и в режиме ее дообучения в процессе эксплуатации. Ее задачей, в частности, является преобразование знаний в форму, позволяющую машине логических выводов (МЛВ) использовать их в процессе работы.

*Рабочая память* или *база данных* (БД) хранит факты о текущем состоянии предметной области, промежуточных и окончательных результатах вывода.

*База знаний* служит для хранения знаний о проблемной области. Форма хранения соответствует выбранной модели представления знаний.

*Машина логических выводов* (МЛВ) или *интерпретатор* осуществляет вывод решения задачи на основе имеющихся в системе знаний и фактов. БД, БЗ и МЛВ составляют ядро ЭС.

*Подсистема объяснения* обеспечивает трассировку хода вывода решения и предоставление по требованию пользователя объяснения вывода с нужной степенью детализации. Эта функция исключительно важна для ЭС, т.к. при принятии ответственных решений на основе рекомендаций ЭС пользователь, как правило, желает знать, каким образом они были получены.

Этапы разработки и стадии жизненного цикла ЭС.

В процессе разработки ЭС принято выделять пять взаимодействующих и частично пересекающихся этапов: идентификация, концептуализация, формализация, реализация и тестирование.

Этап *идентификации* – происходит осмысление необходимости решения задачи методами инженерии знаний, уточняются цели и задачи ЭС, определяются участники процесса разработки и их роли, а также требуемые ресурсы, в том числе возможные источники знаний.

Этап *концептуализации* – строится концептуальная модель проблемной области, т. е. выделяются ключевые понятия, свойства и отношения, необходимые для описания процесса решения задачи. Задача инженера по знаниям на этом этапе состоит в том, чтобы определить, достаточно ли выделенных ключевых понятий и отношений для описания всех имеющихся примеров.

Этап *формализации* – построенная концептуальная модель представляется с использованием выбранных формальных моделей представления знаний. На этом же этапе принимается решение о выборе инструментальных программных средств проектирования ЭС, либо о разработке своих собственных.

Этап *реализации* – создается один или несколько прототипов ЭС, решающих требуемые задачи. На этом этапе выбираются структуры данных и реализуются правила вывода и управляющие стратегии, принятые на этапе формализации, устраняются несоответствия между спецификациями структур данных, правил и схем управления.

Этап *тестирования* – осуществляется оценка работы программы-прототипа на различных входных воздействиях, выявляются ситуации неадекватных решений и определяются их причины.

По степени проработанности и отлаженности в жизненном цикле ЭС выделяют пять стадий: демонстрационный прототип, исследовательский прототип, действующий прототип, промышленная система и коммерческая система.

*Демонстрационный прототип* решает часть требуемых задач, подтверждая потенциальную применимость методов инженерии знаний и принципиальную осуществимость разработки.

*Исследовательский прототип* решает все требуемые задачи, но не полностью отлажен и неустойчив в работе.

*Действующий прототип* надежно решает все задачи, но для решение сложных задач может потребоваться чрезмерно много времени и/или памяти.

*Промышленная система* обеспечивает высокое качество, надежность, быстродействие и эффективность работы в реальных условиях эксплуатации.

*Коммерческая система* пригодна для продажи различным потребителям.

**Лекция 5. Логический вывод в системеCLIPS.**

**Стратегии разрешения конфликтов.**

Базовый цикл работы МЛВ в системеCLIPS:

1. Работа МЛВ останавливается, если достигнут предел активации правил или нет текущего фокуса. В противном случае, для выполнения выбирается верхнее правило агенды того модуля, которому принадлежит текущий фокус. Если в этой агенде нет правил, текущий фокус извлекается из стека фокусов и управление переходит к следующему модулю. Если стек фокусов пуст, то выполнение останавливается, в противном случае вновь выполняется шаг 1.

2. Выполняются операторы, содержащиеся в консеквенте выбранного правила. Использование в консеквенте правила функции return не может переместить текущий фокус из стека фокусов. Счетчик числа правил инкрементируется для контроля предела активации правил.

3. Результатом шага 2 может быть активация или деактивация правил. Активированные правила (у которых удовлетворяется условия в антецеденте) помещаются в агенду того модуля, в котором они определены. Размещение в агенде определяется значимостью (salience) правила и текущей стратегией разрешения конфликтов. Деактивированные правила удаляются из агенды.

4. При использовании динамических значимостей, значения значимостей всех правил, содержащихся в агенде переоцениваются. Цикл повторяется с шага 1.

*Агенда* представляет собой список правил, условия которых удовлетворяются, но которые еще не были выполнены. Каждый модуль имеет собственную агенду. При извлечении правил агенда работает аналогично стеку – первым выполняется правило, находящееся на вершине агенды. Когда правило становится активным (условия в его левой части удовлетворяются), оно помещается в агенду в соответствии со следующими правилами:

1. Вновь активируемые правила помещаются над всеми правилами с более низкой *значимостью* (salience) и ниже всех правил с более высокой значимостью.

2. Для определения места среди правил равной значимости используется текущая стратегия разрешения конфликта.

3. Если в результате добавления или удаления факта одновременно активизируются несколько правил и шаги 1 и 2 не позволяют выполнить упорядочение, то эти правила упорядочиваются между собой произвольно.

*Значимость* позволяет пользователю назначать правилу приоритет, который учитывается при его помещении в агенду. Первым выбирается и срабатывает правило с максимальной значимостью. Значимость может принимать целое значение в диапазоне от -10000 до +10000. По умолчанию ее значение равно 0. Для явного назначения правилу значимости используется оператор:

(declare <rule- salience>),

который добавляется в левую часть правила и размещается перед первым условным элементом, например:

(defrule r1

(declare (salience 500))

(fire test-1)

=>

(printout t "Rule r1 firing." crlf))

Значение значимости может назначаться в один из трех моментов: при определении правила, при активизации правила и в каждом цикле выполнения (последние два случая соответствуют *динамической значимости*). По умолчанию значение значимости назначается только при определении правила. Для изменения такого поведения может использоваться команда set-salience-evaluation.

*Стратегии разрешения конфликтов.* В CLIPS поддерживается семь стратегий разрешения конфликтов: *“вглубь”* (depth), *“вширь”* (breadth), *“простоты”* (simplicity), *“сложности”* (complexity), *LEX*, *MEA* и *случайного выбора* (random).

Текущая стратегия может быть установлена, используя пункт меню “Execution/Options” оконного интерфейса, или командой set-strategy. При этом агенда переупорядочивается на основе новой стратегии. Синтаксис команды:

(set-strategy <strategy>),

где <strategy>::= depth⏐breadth⏐simplicity⏐complexity⏐ lex⏐mea⏐random.

По умолчанию используется стратегия depth.

*Стратегия “вглубь”.* Вновь активируемые правила помещаются в агенду над всеми правилами такой же значимости. Например, пусть факт f-1 активирует правила rule-1 и rule-2, а факт f-2 активирует правила rule-3 и rule-4. Тогда если f-1 устанавливается раньше, чем f-2, то rule-3 и rule-4 окажутся в агенде выше правил rule-1 и rule-2. Однако положение правила rule-1 относительно правила rule-2 и правила rule-3 относительно правила rule-4 будет произвольным.

*Стратегия “вширь”.* Вновь активируемые правила помещаются ниже всех правил с такой же значимостью. Например, пусть факт f-1 активирует правила rule-1 и rule-2, а факт f-2 активирует правила rule-3 и rule-4. Тогда, если f-1 устанавливается раньше, чем f-2, то rule-1 и rule-2 окажутся в агенде выше правил rule-3 и rule-4. Однако, положение правила rule-1 относительно правила rule-2 и правила rule-3 относительно правила rule-4 будет произвольным.

*Стратегия “простоты”.* Среди правил одинаковой значимости, вновь активируемые правила помещаются над всеми правилами с равной или большей специфичностью (specificity). *Специфичность* правила определяется числом сравнений, которые должны быть выполнены в левой части правила.

Специфичность инкрементируется при:

* каждом сравнении с константой или предварительно связанной переменной;
* каждом вызове функции, сделанном из левой части правила в условном элементе с предикатным ограничением (:), ограничением возвращаемым значением (=) или УЭ-проверкой (test).

Булевы функции “и”, “или”, “не” не увеличивают специфичность правила, но их аргументы увеличивают. Вызовы функций, выполняемые из функций, не увеличивают специфичность. Например, следующее правило:

(defrule example

(item ?x ?y ?x)

(test (and(numberp ?x) (> ?x (+ 10 ?y)) (< ?x 100)))

=>…)

имеет специфичность 5 (считаются операторы (item ?x ?y ?x), ?x, numberp, >, <).

*Стратегия “сложности”.* Среди правил одинаковой значимости, вновь активируемые правила помещаются над всеми правилами с равной или меньшей специфичностью.

*Стратегия LEX.* Для определения места правила в агенде среди правил одинаковой значимости в первую очередь используется *новизна* образцов, активирующих данное правило. Каждый факт и экземпляр в БД помечаются “временным тегом”, указывающим его новизну по отношению ко всем другим фактам и экземплярам в системе. Для определения местоположения правила в агенде образцы (факты или экземпляры), связанные с активацией каждого правила сортируются по убыванию новизны. Правило с более поздним образцом помещается выше правил с более ранними образцами. Чтобы определить относительный порядок размещения двух правил, отсортированные временные теги образцов, активирующих правила, сравниваются попарно, начиная с самых больших значений. Сравнение продолжается до тех пор, пока не будет обнаружено, что временной тег одной активации больше соответствующего временного тега другой активации. Правило с большим значением временного тега помещается в агенду выше другого правила. Данная стратегия поясняется примером на рис. 16.1. В данном случае раньше сработает правило Rule1, т.к. временной тег образца, связанного с его третьим условным элементом (8) больше, чем временной тег соответствующего образца у правила Rule2 (6).

Если одно правило имеет больше образцов, чем другое, а все сравниваемые временные теги идентичны, то правило с большим числом временных тегов помещается выше. Если два правила имеют равную новизну, правило с более высокой специфичностью помещается выше правила с более низкой специфичностью.



*Стратегия MEA.* Для определения места правила в агенде среди правил равной значимости в первую очередь используется временной тег образца, связанного с первым условным элементом в правиле. Правило, у которого этот временной тег больше временных тегов, связанных с первыми условными элементами других правил, помещается в агенду выше них. Если временные теги первых образцов равны, то для определения места правила используется стратегия LEX. В примере на рис. 16.2 раньше сработает правило Rule1.

*Стратегия случайного выбора*. Каждому правилу сопоставляется случайное число, которое используется для определения его местоположения в агенде среди правил равной значимости. Это случайное число сохраняется при изменении стратегии, а в случае возврата к случайной стратегии разрешения конфликта восстанавливается тот же порядок среди правил, которые находились в агенде, когда стратегия была изменена.



**Пример построения ЭС в среде CLIPS**

Ниже в качестве примера приведен фрагмент простой ЭС в среде CLIPS, определяющей наличие у студента свободного времени в зависимости от количества несделанных лабораторных работ и оставшегося до зачета времени:

(defrule data-input

(initial-fact)

=>

(printout t crlf "Введите число дней до зачета (целое значение): ")

(bind ?days (read))

(if (numberp ?days)

then (assert (days ?days))

else (printout t "Введите число" crlf))

(printout t crlf "Введите число несделанных лабораторных работ (в %)")

(bind ?works (read))

(assert (works ?works)))

(defrule R1

(days ?days)

(works ?works)

(test (and (= ?days 1) (<> ?works 0)))

=>

(printout t crlf "Свободного времени нет" crlf)

(assert (freetime "no")))

(defrule R2

(days ?days)

(works ?works)

(test (and (= ?days 2) (>= ?works 10)))

=>

(printout t crlf crlf "Свободного времени нет" crlf)

(assert (freetime "no")))

Правило data-input обеспечивает ввод исходных значений числа оставшихся до зачета дней и количества несделанных работ. Оно инициируется исходным фактом (initial-fact) в рабочей памяти и выдает приглашение на ввод значения. Команда bind осуществляет занесение введенных значений в факты рабочей памяти. Кроме того, для переменной выполняется контроль типа введенного значения.

**Лекция 6. Модели представления знаний.**

**Синтаксис и семантика логики предикатов первого порядка**

**Модели представления знаний**

Знания, хранящиеся в базе знаний (БЗ) интеллектуальных систем должны быть представлены с использованием некоторой модели представления знаний.



В настоящее время в теории ИИ принято выделять 4 основных класса моделей представления знаний:

1. Логические модели

2. Продукционные системы

3. Семантические сети

4. Фреймы

Основная цель представления знаний в ИИ – научиться хранить знания таким образом, чтобы программы могли обрабатывать их и достигнуть подобия человеческого интеллекта. Исследователи ИИ используют теории представления знаний из когнитологии (науке о мышлении). Такие методы как фреймы, правила и семантические сети пришли в ИИ из теорий обработки информации человеком. Так как знание используется для достижения разумного поведения, фундаментальной целью дисциплины представления знаний является поиск таких способов представления, которые делают возможным процесс логического вывода, то есть создание выводов из знаний.

В современных системах искусственного интеллекта, как правило, используется комбинация различных моделей. Кроме того, знания принято разделять на *процедурные* и *декларативные*.

Процедурные знания представляют собой описание некоторой упорядоченной последовательности действий, то есть алгоритм.

Декларативные знания, в отличие от процедурных, не предписывают в явной форме каких-либо действий, а представляют собой утверждения о наличии у объектов некоторых свойств или отношений между объектами. Таким образом, декларативные знания носят констатирующий характер.

**Логические модели представления знаний**

"Логика" в переводе с греческого означает "наука о рассуждении", "искусство рассуждения". Существует несколько определений понятия логики:

* Наука о формах, методах и законах интеллектуальной познавательной деятельности, формализуемых с помощью логического языка.
* Наука о достижении истины в процессе познания с помощью выводного знания — знания, полученного опосредованным путём, посредством не чувственного опыта, а из знаний, полученных ранее; знания, полученного разумом.
* Наука о законах мышления (дискос об окружающем мире).

Знание, полученное с помощью применения законов логики и методов логического мышления, является целью любого логического действия, нацеленного на достижение истины. Полученные знания также могут применяться для более глубокого познания явлений и событий окружающего мира.

Основной же функцией логики является исследование того, как из одних утверждений можно выводить другие. При этом предполагается, что вывод зависит только от способа связи входящих в него утверждений и их строения, а не от их конкретного содержания. Другими словами, изучая, "что из чего следует", логика выявляет наиболее общие или, как говорят, формальные условия правильного мышления.

Наиболее известной и широко используемой для представления знаний моделью является модель логики предикатов первого порядка.

***Основные определения***

Рассмотрим основные определения логики предикатов первого порядка.

Под *областью интерпретации D* (от слова Domaine) подразумевается множество объектов, свойства и отношения которых предполагается описывать средствами языка логики предикатов первого порядка.

Язык логики предикатов первого порядка (ЛП1) строится следующим образом: алфавит ЛП1 включает в себя следующие группы символов:

1. Предметные константы:

*a, b, c, d,…*

2. Предметные переменные:

*x, y, z, v, u, w,… xn, yn, zn,…*

3. Функциональные символы:

*f, g, h,…*

4. Предикатные символы:

*P, Q, R, S, T,…*

5. Логические связки:

*¬, v, &, →, ↔*

6. Логические кванторы:

 (квантор существования), (квантор всеобщности)

7. Скобки:

*(, ),…*

*Предметные константы* – идентификаторы конкретных объектов в рассматриваемой предметной области. Предметные константы выступают в качестве имен (идентификаторов) элементов области интерпретации D.

*Переменные* – то, что может принимать значения констант.

*Функциональная форма* – соответствует функции, заданной на предметной области:

Dn → D (отображение декартового произведения в область интерпретации)

Синтаксически задается с помощью функционального символа и списка аргументов: f(f1,…,fn). Примеры:

1). *f(x,y) – "x+y"*

2). *g(x) – "x2"*

3). *f(a, x, g(x))*

и так далее.

*Термом* называется предметная константа, предметная переменная или функциональная форма.

*Предикатные символы* – служат для обозначения свойств и отношений объектов в рассматриваемой предметной области (области интерпретации).

Число предметных переменных, к которым относится данная предикатная форма, называется ее *местностью*.

Пример:

*P(x)* – "x – четное число" – одноместная предикатная форма

*Q(x,y)* – "x > y" – двуместная предикатная форма

*Логические связки* – имеют традиционный смысл:

→ – импликация (если . . . , то . . . )

↔ – эквивалентность

*Логические кванторы*:

 – "для всех x из области P"

 – "существует хотя бы один объект из области P"

***Правила построения формул в логике предикатов***

1). Любая предикатная форма (или атом) является формулой логики предикатов первого порядка.

2). Если X и Y – формулы, то:

* и т.д.* – тоже формулы

Бывают и невыполнимые формулы, например:

 – невыполнимая.

3). Если x – предикатная переменная, А – формула, тогда:

 и  – также формулы.

4). Других формул нет.

Формула, которой предшествует квантификатор, называется *областью действия* этого квантификатора.

Если переменная находится в области действия соответствующего квантификатора, то она называется *связанной*, в противном случае – *свободной*. Например: ,

*x* – связанная переменная

*y* – свободная.

Если формула не имеет свободных переменных, то она является *замкнутой* и является *высказыванием*.

Рассмотрим пример: для любых двух чисел, если одно из них четно, а другое – нечетно, то их сумма нечетна.

 – зафиксировали область интерпретации.

 – "x – четно"

 – "x + y"

 – истинное высказывание.

Логика, в которой рассматриваются только высказывания об объектах, свойствах и отношениях предметной области, называется *логикой первого порядка*. Если рассматриваются высказывания о высказываниях – имеем дело с *логикой второго порядка* и т. д.

**Лекция 7. Логическое следование. Логический вывод.**

**Метод резолюций в логике предикатов первого порядка.**

Логика первого порядка, являясь формализованным аналогом обычной логики, дает возможность строго рассуждать об истинности и ложности утверждений и об их взаимосвязи, в частности о логическом следовании одного утверждения из другого, или, например, об их эквивалентности.

Рассмотрим классический пример формализации утверждений естественного языка в логике первого порядка. Возьмем рассуждение: "Каждый человек смертен. Конфуций – человек. Следовательно, Конфуций смертен". Обозначим:

"x есть человек" – через *ЧЕЛОВЕК(x),*

"x смертен" – через *СМЕРТЕН(x).*

Тогда утверждение "каждый человек смертен" может быть представлено формулой:

*()(ЧЕЛОВЕК(x) → СМЕРТЕН(x)).*

А утверждение "Конфуций – человек" – формулой:

*ЧЕЛОВЕК(Конфуций),*

и "Конфуций смертен" – формулой:

*СМЕРТЕН(Конфуций).*

Утверждение в целом теперь может быть записано формулой:

*()(ЧЕЛОВЕК(x) → СМЕРТЕН(x)) & ЧЕЛОВЕК(Конфуций) → СМЕРТЕН(Конфуций).*

***Понятие логического следования***

Пусть *E* – множество формул (логики первого порядка), а *C* – отдельная формула.

Формула *C* называется *логическим следствием* из множества формул *E*, если она истинна при всех интерпретациях, при которых *все формулы* множества *E одновременно истинны.*

Формальная запись:

*E* ╞  *C*

Используя данное определение факт логического следования можно установить путем перебора всех возможных интерпретаций. Однако, в логике предикатов такой подход невозможен, т. к. каждая формула имеет бесконечное число интерпретаций (вследствие бесконечного множества областей интерпретаций).

***Принцип дедукции***

Множество формул называется невыполнимым, если не существует интерпретации, при которой все формулы этого множества одновременно истинны:

*E* ╞ *C* ⬄ – невыполнимо.

Таким образом, принцип дедукции сводит задачу о логическом следовании к задаче о невыполнимости множества формул.

Для доказательства невыполнимости множества формул используется метод резолюции.

***Метод резолюции***

Если *X* и *Y* – дизъюнкты, то формула  называется *резольвентой*.

Правило резолюции: пусть *x, y* – произвольные формулы, а *A* – атом (элементарная формула), тогда:

╞ – правило резолюции.

, , тогда 

Предположим, что , тогда . Таким образом .

Рассмотрим случай, когда x и y – дизъюнкты (дизъюнкция атомов).

Любая формула может быть преобразована в логически эквивалентную ей КНФ (конъюнктивно-нормальную форму).

Дизъюнкт, содержащий хотя бы одну литеру, является выполнимой формулой (то есть при неких интерпретациях является истинной).

Единственным невыполнимым дизъюнктом является пустой дизъюнкт, то есть дизъюнкт, не содержащий ни одной литеры. Такие дизъюнкты будем обозначать в дальнейшем константой F (от False).

***Алгоритм по преобразованию формул в КНФ (на примере логики высказываний)***

1). Исключение связок, импликаций и эквивалентностей:





2). Сокращение области действия отрицаний так, чтобы они относились к элементарным формулам:



3). Закон дистрибутивности:



Утверждение 1: пусть *E* – множество формул,  – формулы этого множества, а *{Y,X}*╞ *Z* –логическое следствие*.* Тогда добавление формулы *Z* к множеству *E* не меняет его выполнимости/невыполнимости.

Утверждение 2: добавление резольвент к исходному множеству не меняет его выполнимости/невыполнимости.

Для построения резольвенты находим дизъюнкты, содержащие контрарную пару (пара, имеющая противоположные значения) и строим резольвенту.

Пример 1:

  – родительские дизъюнкты,

 – их резольвента.

Пример 2: рассмотрим 2 однолитеральных дизъюнкта  и , их резольвента – пустой дизъюнкт *F*.

Таким образом, алгоритм метода резолюции предполагает последовательное порождение резольвент от различных пар родительских дизъюнктов до тех пор, пока не будет получен пустой дизъюнкт. Это означает, что множество невыполнимо и тем самым логическое следствие доказано.

Если пустой дизъюнкт получить не удастся, то множество является выполнимым и значит, что логическое следствие не имеет места.

Теорема Робинсона: если множество дизъюнктов невыполнимо, этот факт всегда может быть установлен с помощью метода резолюции за конечное число шагов. Можно говорить, что метод резолюции полон для данной проблемы.

***Общие достоинства логики первого порядка***

Логика первого порядка обладает рядом полезных свойств, которые делают ее очень привлекательной в качестве одного из основных инструментов формализации знаний:

1. Полнота.

2. Непротиворечивость.

3. Компактность.

**Лекция 8. Продукционные системы. Управление выводом в продукционных системах.**

***Основные определения***

В самом общем виде продукционная система задается с помощью совокупности правил следующего вида:

*Если S1, то R1, иначе…*

*…*

*Если Sn, то Rm,*

где:

*S* – описание некоторой ситуации,

*R* – совокупность действий, которые должны быть выполнены в этой ситуации.

Левая часть правила называется *антецедентом* (LHS – Left Hands Side), правая – *консеквентом* (RHS – Right Hands Side).

Продукционная система обеспечивает управление процессом решения задачи на основе сопоставления с образцом. Структурно система состоит из набора продукционных правил, рабочей памяти и цикла управления "распознавание-действие".

*Набор продукционных правил* (production rule) или продукций (productions).

*Продукция* – это пара "условие-действие", которая определяет одну порцию знаний, необходимых для решения задачи.

*Условная часть правила* – это образец (шаблон), который определяет, когда это правило может быть применено для решения какого-либо этапа задачи.

*Часть действия* – определяет соответствующий шаг решения задачи.

*Рабочая память* (working memory) – содержит описание текущего состояния мира в процессе рассуждений. Это описание является образцом, который сопоставляется с условной частью продукции с целью выбора соответствующих действий при решении задачи. Если условие некоторого правила соответствует содержимому рабочей памяти, то может выполняться действие, связанное с этим условием. Действия продукционных правил предназначены для изменения содержания рабочей памяти.

*Конфликтное множество* (conflict set) или *агенда* – совокупность правил, готовых к выполнению.

**Логический вывод в продукционных системах**

Существуют два основных метода вывода в продукционных системах: прямой и обратный.

При *обратном выводе* система начинает решение с попытки определить значение целевой переменной. Для этого просматривается набор правил и ищется правило, с помощью которого можно достичь требуемой цели (присвоить целевой переменной конкретное значение). После того, как такое правило будет найдено, проверяется истинность его антецедента. Предикаты, содержащиеся в посылках, становятся текущей целью (подцелью) и вывод продолжается рекурсивно. Обратный вывод использовался в языке "Prolog".

При использовании *прямого вывода* обработка выполняется от посылок к заключению. По этому алгоритму работает интерпретатор машины логических выводов (МЛВ).

***Алгоритм сопоставления с образцами***

Процедура сопоставления сопоставляет условия (предикаты), содержащиеся в антецеденте правил в базе знаний (БЗ) с фактами, хранящимися в базе данных (БД) (см. рис).



Правила, для которых сопоставление прошло успешно, помещаются в агенду (конфликтное множество – КМ). Далее на основе правил разрешения конфликтов (ПРК) выбирается одно активизируемое правило (АП). Это правило срабатывает, в результате чего в БД добавляются новые факты, и цикл работы МЛВ повторяется.

Ряд стратегий разрешения конфликтов реализованы в среде CLIPS. Известные стратегии:

1. Принцип "стопки книг". Основан на идее, что наиболее часто используемая продукция является и наиболее полезной. Правила в агенде упорядочиваются по частоте их использования в прошлом.

2. Принцип метапродукций. В систему добавляются специальные правила – метапродукции, предназначенные для организации управления в процессе выборки. Например, в MYCIN:

*Если Инфекция есть* ***pel\_abc*** *и в агенде имеются правила,*

*у которых в условии A упоминается* ***gramm\_r****,*

*то продукции, у которых в A имеются* ***ent…***

*следует активизировать раньше,*

*чем продукции, содержащиеся в A* ***gramm\_r****.*

***Модификации продукционной модели***

Продукционная модель часто дополняется определённым порядком, вводимым на множестве продукций, что упрощает механизм логического вывода. Порядок может выражаться в том, что отдельная следующая по порядку продукция может применяться только после попыток применения предшествующих ей продукций. Примерно похожее влияние на продукционную модель может оказать использование приоритетов продукций, означающее, что в первую очередь должна применяться продукция, имеющая наивысший приоритет.

Рост противоречивости продукционной модели может быть ограничен путём введения механизмов исключений и возвратов. Механизм исключений означает, что вводятся специальные правила-исключения. Их отличает большая конкретность в сравнении с обобщёнными правилами. При наличии исключения основное правило не применяется. Механизм возвратов же означает, что логический вывод может продолжаться в том случае, если на каком-то этапе вывод привёл к противоречию. Просто необходимо отказаться от одного из принятых ранее утверждений и осуществить возврат к предыдущему состоянию.

***Общие достоинства продукционных систем***

1. Модульность организации знаний.

2. Независимость правил, выражающих самостоятельные фрагменты знаний.

3. Простота и естественность модификаций знаний в БЗ.

4. Отделение предметных знаний от управляющих, что позволяет применять различные управляющие стратегии и создавать общие управляющие механизмы для разных приложений.

***Основной недостаток продукционных систем***

Продукционная модель представления знаний обладает тем недостатком, что при накоплении достаточно большого числа (порядка нескольких сотен) продукций они могут друг другу противоречить.

**Лекция 9. Фреймы, как модель представления знаний.**

**Управление выводом во фреймовых системах.**

*Фрейм* (от англ. Frame – рамка, несущая конструкция) – структура для описания стереотипной ситуации, содержащая множество характеристик ситуации, называемых *слотами*, и их значений.

Знания о мире могут быть представлены в виде фрейм-подобных структур. При анализе ситуации люди, как правило, используют свой опыт и адаптируют эти знания применительно к новой ситуации. Например, человек однажды останавливавшийся в гостинице, имеет представление о всех гостиничных номерах. Они содержат кровать, шкаф, ванную и т. д. Детали каждого номера (цвет портьер, расположение и тип выключателей и т. п.) могут отличаться. С фреймом гостиничного номера связана также информация, принимаемая по умолчанию. Например «Если нет простыней - нужно вызвать горничную», «если нужен лед - необходимо посмотреть в холле», и т. п. Не надо иметь знания каждого нового гостиничного номера. Все элементы обобщенного номера организуются в концептуальную структуру, к которой человек обращается, когда останавливается в гостинице.

Эти высокоуровневые структуры можно представить в семантической сети, организуя ее как совокупность отдельных сетей, каждая из которых представляет некоторую стереотипную ситуацию. Фреймы так же, как *объектно-ориентированные системы,* обеспечивают механизм подобной организации, представляя сущности, как структурированные объекты с поименованными ячейками и связанными с ними значениями. Таким образом, фрейм или схема рассматривается как единая сложная сущность.

Общий вид фреймовой структуры:

(*Имя\_фрейма, (Имя\_слота\_1, Значение\_слота\_1),*

*(Имя\_слота\_2, Значение\_слота\_2),*

*… … …*

*(Имя\_слота\_N, Значение\_слота\_N)*

)

Например, гостиничный номер и его компоненты могут быть описаны рядом отдельных фреймов:

Помимо кровати во фрейме должен быть представлен:

* стул: ожидаемая высота — от 20 до 40 см, число ножек — 4, значение по умолчанию — предназначен для сидения.
* гостиничный телефон: это вариант обычного теле­фона, но расчет за переговоры связывается с оплатой гостиничного номера. По умолча­нию существует специальный гостиничный коммутатор, и человек может использовать этот телефон для заказа еды в номер, внешних звонков и получения других услуг.

На рис. 11.1 изображен фрейм, представляющий гостиничный номер:



Рис. 11.1 – Фрейм гостиничного номера.

Как видно из примера, фрейм может содержать любое число слотов, в конкретных системах некоторое число слотов м.б. обязательными, другие – необязательными.

Подробная структура фрейма:

(*Имя\_фрейма,* (*Имя\_слота\_1,*

*Указатель\_наследования,*

*Указатель\_типа\_данных,*

*Значение\_слота,*

*Демон*

)*,*

*… … …*

(*Имя\_слота\_N,*

*Указатель\_наследования,*

*Указатель\_типа\_данных,*

*Значение\_слота,*

*Демон*

)*,*

)

**Имя\_фрейма** – идентификатор, уникальный в данной фреймовой системе

**Имя\_слота\_X** – уникальный идентификатор слота в пределах данного фрейма

**Указатель\_наследования** – используется во фреймовых системах иерархического типа, позволяющих организовать иерархию знаний. Показывает, какую информацию об атрибутах слотов в фрейме верхнего уровня наследуют слоты с теми же именами нижнего уровня.

В сети каждое понятие представляется узлами и связями на одном и том же уровне детализации. Однако очень часто для одних целей объект необходимо рассматривать как единую сущность, а для других — учитывать детали его внутренней структуры. Например, обычно нас не интересует меха­ническое устройство автомобиля, пока что-то не сломается. При обнаружении поломки мы достаем схему автомобильного двигателя и пытаемся устранить проблему.

Системы фреймов поддерживают наследование классов. Значения ячеек и используе­мые по умолчанию значения класса наследуются через иерархию класс/подкласс и класс/член. Например, гостиничный телефон можно реализовать как подкласс обычного телефона, обладающий следующими особенностями: внешние звонки проходят через коммутатор гостиницы (для расчета), и гостиничные услуги можно заказывать по пря­мому номеру.

**Типовые указатели наследования:**

**U** – unique, «Уникальный», указывает, что каждый фрейм может иметь слоты с различными, уникальными значениями.

Фрейм высшего уровня

|  |  |  |
| --- | --- | --- |
| **Фрейм** | **Слот** | **Значение** |
| Человек | Масса | 50 |

Фрейм среднего уровня

|  |  |  |
| --- | --- | --- |
| **Фрейм** | **Слот** | **Значение** |
| Ребенок | Масса | 25 |

Фрейм низшего уровня

|  |  |  |
| --- | --- | --- |
| **Фрейм** | **Слот** | **Значение** |
| Петя | Масса | 32 |

Как видно в примере, каждый слот имеет свое собственное независимое значение.

**S** – same, «такой же», - все слоты должны иметь одинаковые значения

Фрейм верхнего уровня

|  |  |  |
| --- | --- | --- |
| **Фрейм** | **Слот** | **Значение** |
| Человек | Масса | 50 |

Фрейм среднего уровня

|  |  |  |
| --- | --- | --- |
| **Фрейм** | **Слот** | **Значение** |
| Ребенок | Масса | 50 |

Фрейм низшего уровня

|  |  |  |
| --- | --- | --- |
| **Фрейм** | **Слот** | **Значение** |
| Петя | Масса | 50 |

Тут все фреймы в иерархии имеют одинаковое значение.

**R** – range – значение слотов фрейма должны находится в диапазоне указанных значений слотов фрейма верхнего уровня

Фрейм верхнего уровня

|  |  |  |
| --- | --- | --- |
| **Фрейм** | **Слот** | **Значение** |
| Человек | Масса | 30 -- 150 |

Фрейм среднего уровня

|  |  |  |
| --- | --- | --- |
| **Фрейм** | **Слот** | **Значение** |
| Ребенок | Масса | 10 -- 40 |

Фрейм верхнего уровня

|  |  |  |
| --- | --- | --- |
| **Фрейм** | **Слот** | **Значение** |
| Петя | Масса | 32 |

Значение слота последующего фрейма находится в диапазоне, определенном «родителем»

**O** – override, «Игнорировать» - при отсутствии указания, значение слота верхнего уровня наследуется слотом нижнего уровня, а в случае явного определения нового значения наследование не используется.

1 - Фрейм высшего уровня

|  |  |  |
| --- | --- | --- |
| **Фрейм** | **Слот** | **Значение** |
| Человек | Масса | 70 |

2 - Фрейм среднего уровня

|  |  |  |
| --- | --- | --- |
| **Фрейм** | **Слот** | **Значение** |
| Ребенок | Масса | 50 |

3 - Фрейм низшего уровня

|  |  |  |
| --- | --- | --- |
| **Фрейм** | **Слот** | **Значение** |
| Петя | Масса | 50 |

Во втором фрейме значение слота было переопределено, а в третьем – осталось по умолчанию.

**Указатель\_типа\_данных**: аналог типизации переменных в программировании.

Возможные типы данных:

* FRAME – указатель на другой фрейм
* INT
* REAL
* BOOL
* TEXT
* LIST
* TABLE
* EXPR

**Значение\_слота** - тип значения должен совпадать с явно указанным.

**Демон** – процедура, автоматически запускаемая при выполнении некоторого условия.

Демоны запускаются при обращении к соответствующему слоту и выполняют определенные действия со значением слота.

|  |  |
| --- | --- |
| Типичные демоны | |
| IF-NEEDED | Запускается, если в момент обращения к слоту его значение не установлено |
| IF-ADDED | запускается при записи в слот некоторого значения |
| IF-REMOVED | запускается при удалении значения слота |

Демоны, по сути, являются разновидностями *присоединенных процедур* – программ, которые используются для определения значения слота. Например, при каждом изменении определенной ячейки в системе можно запускать процедуру контроля соответствия типов или тест непроти­воречивости.

Подобные процедурные вложения являются важным свойством фреймов, так как они по­зволяют связать фрагменты программного кода с соответствующими сущностями фреймового представления. Например, в базу знаний можно включить возможность генерировать графические образы.

Присоединенные процедуры запускаются по сообщениям, переданным из другого фрейма.

Демоны и присоединенные процедуры представляют собой *процедурные знания* в составе фреймов. Остальные знания в данной системе – декларативные.

Фреймы расширяют возможности семантических сетей, позволяя представлять слож­ные объекты не в виде большой семантической структуры, а ввиде единой сущности (фрейма). Это также позволяет естественным образом представить стереотипные сущно­сти, классы, наследование и значения по умолчанию.

**Лекция 10. Семантические сети. Вывод на семантических сетях.**

Формально, *семантическая сеть* (СС) представляет собой помеченный ориентированный граф. Вершины СС соответствуют сущностям предметной области – объектам, событиям, свойствам, процессам, явлениям.



Системы наследования позволяют запоминать информацию на самом высоком уровне абстракции, что уменьшает размер баз знаний и помогает избежать противоре­чии. Наследование также помогает поддерживать непротиворечивость базы знаний при добавлении новых классов и объектов.

С точки зрения математической логики, каждая вершина соответствует элементу предметного множества, а дуга — предикату.

|  |  |
| --- | --- |
| Классификация семантических сетей | |
| По сложности структуры | * Простые сети – вершины не имеют внутренней структуры * Иерархические сети |
| По количеству типов отношений | * Однородные – только один тип отношений * Неоднородные |
| По арности (количеству понятий, связанных одной дугой) | * Бинарные * N-арные |

Пример простой СС:



На языке логики предикатов это будет выглядеть следующим образом:



где: P(x) – “x - ласточка”

Q(x)- “x - птица”

Наиболее часто возникает потребность в описании отношений между элементами, множествами и частями объектов.

Отношение между объектом и множеством, обозначающим, что объект принадлежит этому множеству, называется отношением классификации (IS-A). Говорят, что множество (класс) классифицирует свои экземпляры. Название произошло от английского «IS-A». Иногда это отношение именуют также MemberOf. Отношение IS-A предполагает, что свойства объекта *наследуются* от множества.

Обратное к IS-A отношение используется для обозначения примеров, поэтому часто называется «Example».

Отношение между надмножеством и подмножеством называется AKO — «A KIND OF» («разновидность»). Альтернативные названия — «SubsetOf» и «Подмножество». Это отношение определяет, что каждый элемент первого множества входит и во второе (выполняется IS-A для каждого элемента), а также логическую связь между самими подмножествами: что первое не больше второго и свойства первого *множества* наследуются вторым.

Объект, как правило, состоит из нескольких частей, или элементов. Например, компьютер состоит из системного блока, монитора, клавиатуры, мыши и т. д. Важным отношением является HAS-PART, описывающее части/целые объекты.

В С-сети имеет место иерархия понятий, позволяющая делать выводы на основе наследования.

Наследование в данном случае обусловлено транзитивностью отношений IS-A и HAS-PART:



Часто отношения рассматриваются как сущности, имеющие собственные атрибуты, например отношение владения может характеризоваться периодом владения.



Использование данной сети можно реализовать процедурой знаний, отвечающей на различные вопросы: Чем владеет Юко? В течении какого периода?

**Вывод на семантических сетях**

В основе процедур вывода на семантической сети лежит сопоставление частей сетевой структуры (подграф).

При этом строится подсеть, соответствующая запросу.

*Пример:* Запросу «Чем владеет Юко?» соответствует подсеть:



При обработке запроса проводится сопоставление графа с базой знаний (семантической сетью).

Отыскивается вершина [владеет \*] (или дуга «владелец», направленная к вершине [Юко]). Затем определяется вершина, к которой ведет дуга «объект владения». Метка вершины – ответ на вопрос.

Запросу «Существует ли птица, которая владеет гнездом?» соответствует следующий граф.



Теория семантических сетей получает применение при создании *семантической паутины* – метода представления данных в сети Интернет, при котором возможна их семантическая обработка.

Семантическая паутина — часть концепции развития Всемирной паутины и сети Интернет, принятой консорциумом W3. Целью внедрения этой концепции является создание сети документов, содержащих метаданные, существующей параллельно с ресурсами, предназначенными для восприятия человеком (в данном контексте *ресурсом* является любой объект, доступный в Интернете: веб-страница, изображение, аудио- или видеоматериал, и т. п.), и позволяющей машинам (программным *интеллектуальным агентам*) автоматически производить чёткие (недвусмысленные) заключения о свойствах этих ресурсов.

Техническую часть семантической паутины составляет семейство стандартов на языки описания, включающее XML, XML Schema, RDF, RDF Schema, OWL, а также некоторые другие.

**Достоинства и недостатки семантических сетей**

* Достоинства: Подобная форма - самая удобно воспринимаемая человеком.
* Недостатки: Трудности при построении сложных сетей и попытке учета особенностей естественного языка.

**Лекция 11. Поиск в пространстве состояний.**

**Формальная постановка задачи. Обобщенный алгоритм поиска.**

Многие задачи, в частности игры и головоломки, могут быть представлены как задачи *поиска в пространстве состояний*.

Решить задачу – значит найти путь из исходного состояния в целевое.

Формально задача *поиска в пространстве состояний* в общем случае задается четверкой:

<*I*, {*Oi*}, *GT*, *PC*>,

где *I* – исходное состояние, т. е. состояние мира в начале задачи;

     {*Oi*} – множество действий (операторов), возможных в различных состояниях;

*GT* (*goal test*) – проверка достижения целевого состояния;

*PC* (*path cost*) – функция стоимости пути.

Действия (операторы) переводят состояния в другие состояния. Таким образом, можно ввести в рассмотрение *функцию последователей* *S* (successor), ставящую в соответствие каждому состоянию *x* множество состояний *S*(*x*), достижимых из *x* за одно действие.

Исходное состоянии и множество действий (операторов) в совокупности определяют *пространство состояний задачи*, т.е. множество всех состояний, *достижимых* из исходного, путем конечной последовательности действий.

Проверка достижения целевого состояния позволяет для каждого достигнутого состояния определить является ли оно целевым. Существует два способа задания целевых состояний:

* явное перечисление множества целевых состояний;
* использование предикатов, описывающих целевые состояния.

Как правило, некоторое решение задачи является более предпочтительным, чем другие.

Функция *PC* (path cost) позволяет вычислить стоимость пути в заданных единицах. Как правило *PC* является аддитивной оценкой, т. е. стоимость пути вычисляется как сумма стоимостей элементов пути (операторов).

Для описания алгоритмов реализации поиска в пространстве состояний компоненты задачи должны быть представлены соответствующими данными:

datatype Problem component : INITIAL-STATE, Оperations, Goal Test, Path-Cost-Funct.

Эффективность того или иного *метода поиска* определяется ответами на следующие вопросы:

1. Находит ли он решение в принципе (полнота);
2. Является ли найденное решение хорошим (т.е. имеющим низкую стоимость пути).
3. Какова стоимость реализации поиска, определяемая временем и памятью, требуемыми для нахождения решения (пути).

*Полная стоимость поиска* есть сумма стоимости пути и стоимости поиска пути. Решением задачи – является путь к целевому состоянию, имеющий конкретную стоимость. Чтобы найти этот путь нужно проделать поисковые действия, что требует времени и памяти.

Процесс поиска в пространстве состояний удобно рассматривать как *построение дерева поиска*, которое накладывается на пространство состояний.

Корнем этого дерева является вершина соответствующая начальному состоянию.

Листья дерева соответствуют состоянию, не имеющему потомков в дереве, либо из-за того, что они еще не были раскрыты, либо они сгенерировали пустое множество потомков.

На каждом шаге алгоритм поиска выбирает для раскрытия одну концевую вершину.

Общий алгоритм поиска можно представить следующим образом:

**Function** *General-Search* (Problem, Strategy) **returns** solution or failure

Инициализация дерева поиска исходным состоянием задачи I

**Loop do**  - цикл

**if** (нет вершин - кандидатов для раскрытия)

**then** return failure

Выбрать концевую вершину (лист) для раскрытия в соответствии со стратегией.

**if** (вершина содержит целевое состояние)

**then** return solution (путь к этой вершине)

**else**

Раскрыть вершину и добавить новые вершины в дерево поиска.

**end**

Важно различать *пространство состояний* и *дерево поиска*. Для представления вершин в дереве поиска необходимо использовать структуры данных, имеющие следующие компоненты:

1. Состояние в пространстве состояний, которому сопоставлена вершина.

2. Родительская вершина – это вершина, непосредственным потомком которой является данная вершина.

3. Оператор, в результате применения которого была порождена данная вершина.

4. Глубина вершины – число вершин в пути от корня дерева к данной вершине.

5. Стоимость пути от корневой вершины к текущей.

Таким образом, необходимо различать *вершины* и *состояния* в пространстве поиска.

Состояние представляет собой элемент пространства состояний , т.е. некоторое состояние мира. Вершина – это структура данных, используемая для представления дерева поиска. Таким образом, вершина имеет глубину и родителей, а состояния не имеют.

Множество вершин, ожидающих раскрытия, принято называть *каймой* или границей (fringer).

Стратегия поиска представляет собой функцию, выбирающую из каймы (граничного множества вершин) очередную вершину для раскрытия.

С алгоритмической точки зрения граничное множество удобно представить очередью и определить над ней следующие операции:

1. make-Queue (Elements) – создание очереди с заданными элементами;

2. empty?(Queue) – возвращает true, если очередь пуста;

3. remove-front(Queue) – возвращает первый элемент очереди и удаляет его из очереди;

4. Queueing-Fn(Elements, Queue) – добавляет в очередь множество элементов. Именно эта функция определяет разные стратегии, реализующие разные алгоритмы поиска.

С использованием введенных обозначений можно записать алгоритм поиска:

**function** *General-Search*(Problem, Queuing-Fn) **returns** solution or failure

    nodes 🡨 make-Queue(Make-Node(Init-State[Problem])

//присваивание Make-Node;

// порождение вершины, т.е. получили корень

**loop do**

**if** nodes = 0 **then** return failure

node 🡨 Remove-Front (nodes)

**if** Goal-Test[problem] (State(node))=true

**then** return node

**else** nodes 🡨 Queueing-Fn(nodes, Expand(node, OPERATORS[Problem]))

**end**

Последний оператор к множеству уже существующих вершин добавляет результат раскрытия Expand. OPERATORS обеспечивают переход из одного состояния в другое. Результатом являются все возможные потомки в вершине Expand.