Содержание

1 Лекция 1

– Lisp - безтиповый язык. Атомы - символы, которые могут обозначать любый объекты. Самовычислительные атомы - T, Nil, числа, строки.

Точечная пара - (A.B):
$\square\square \rightarrow cdr$
$\downarrow$
car
Пустая конструкция - () - Nil.

2 Лекция 2

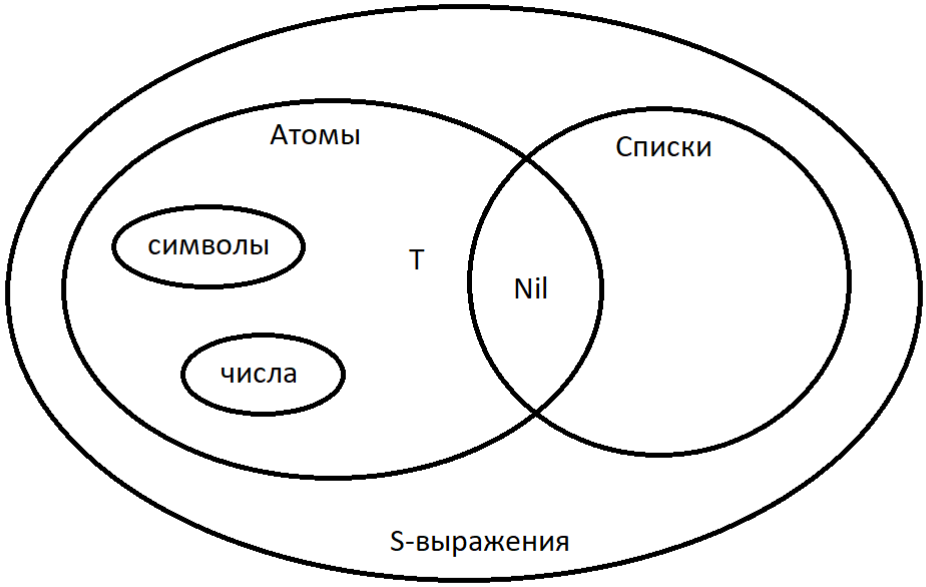

Рисунок 2.1 — Виды структур в lisp

2.1 Представление атома в памяти

Представление атома в памяти - атом представляется в памяти в виде структуры, содержащей пять указателей:

— name - символьный идентификатор атома

— value - самоопределяемое значение атома

— function - лямбда-выражение

— properties - список свойств

— package - пакет, указатель на начало области, связывающий значение атома и пространство имён

2.2 Классификация функций

— чистые функции (математические)

— формы (специальные функции), могут принимать различное число аргументов

— псевдо-функции (вывод на экран)

— функции с вариантами значений

— функционалы - принимают функцию в качестве параметров, либо возвращаемым значением является функция

— базисные функции - car, cdr, cons, atom, cond, quote, eval, lambda, apply, funcall

Выяснить атом - перейти по указателю.
Лямбда-выражение - (lambda (параметры) (тело)).
Вызов: (лямбда-выражение аргументы)

2.3 Именованные функции

Синтаксис:
(defun имя лямбда-выражение)
(defun f (x_1,...x_k) форма)

Лямбда-определение может оказаться более эффективным. Пример лямбда-определения:
$(let(x_1p_1)(x_2p_2)...(x_kp_k)e) = ((lambda(x_1, x_2,...x_k)e)p_1p_2...p_k)$

2.4 Классификация по операциям работы со списку

1. Селекторы - car, cdr
2. Функции-конструкторы - cons, list
3. Функции-предикаты - atom, null, listp, consp, eql (eql сравнивает атомы и числа одного типа, применим только к числам) equal (как eql и списки), equalp (equal + -").

2.5 Вычисление функций

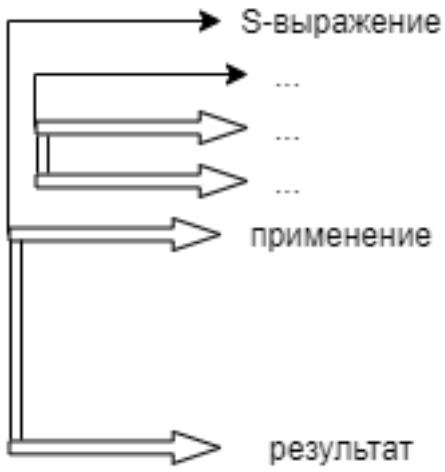

Рисунок 2.2 — Пример построения схемы вычислений

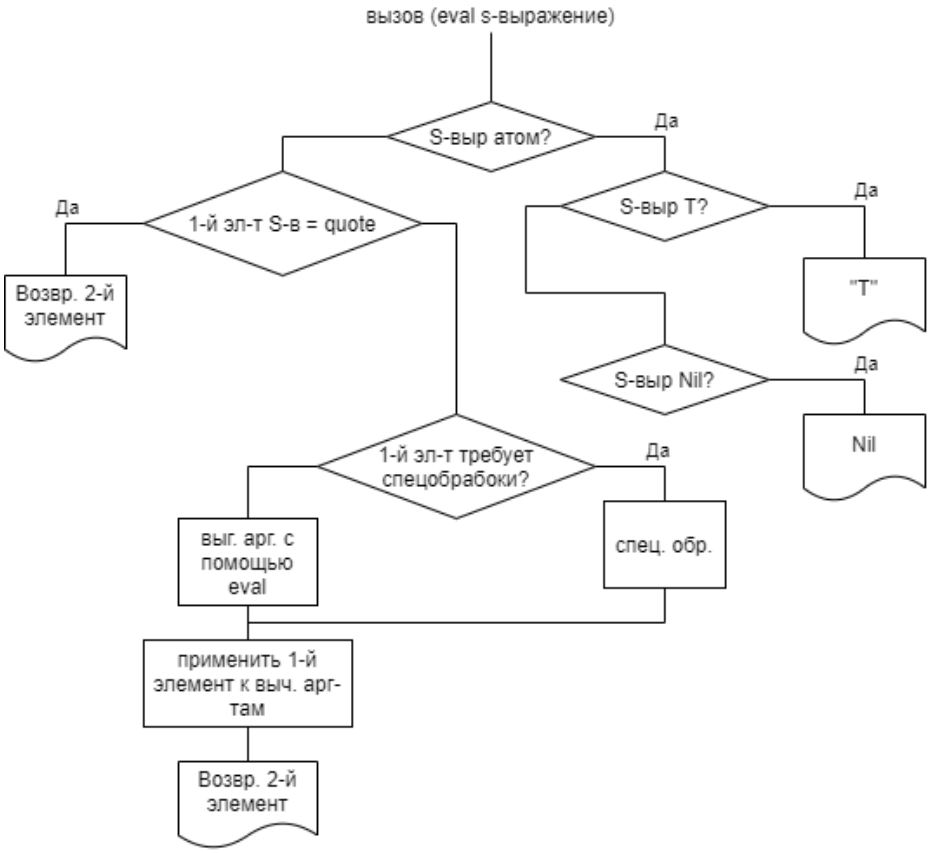

Рисунок 2.3 — Пример построения блок-схемы работы функции

3 Лекция 3

Как функция отличает символы от функций? Как система запоминает символы и на какое время? Придумать метод реализации учитывая, что Lisp реализуется с помощью указателей.

3.1 Функции

Мн-во всех функций можно классифицировать:
1. как базисные и небазисные
2. по методу написания:
- чистые функции имеют фиксированное количество аргументов и возвращают результат
- формы - функция, которая вычисляет не все свои аргументы
- функционалы - принимают на вход функцию или возвращают функцию в качестве результата

3.2 Специальные функции

- базисная функция (cond (test_1 body_1) (test_2 body_2) ... (test_i body_i) [(t body_i+1)]), test_i - тестирующееся выражение, body_i - выполняется, если test_i корректный. - if - более удобная реализация cond, (if test than else)/(if test than) - если нет else, то автоматически возращается nil
Диаграммы во второй лабораторной. При вызове свой функции указываются фактические параметры, в определении функции присутствуют формальные параметры. Если формальный параметр получает какое-то значение, то к этому значению необходимо иметь доступ, поэтому этому символу должна быть выделена память. Для каждого атома - 5 указателей. В процессе работы тела указатели могут переставляться и могут меняться значения, на которые указывают эти указатели. И поэтому на каждом шаге использования этого параметра система вынуждена каждый раз вычислять значение этого атома.
Формальные параметры существуют только во время работы функции -> в рекурсивной функции есть опасность потерять значение.
- логические выражения (формы and/or, not, null)
Символьному атому можно установить значение. Есть локальные атомы - существуют в фукнции, есть глобальные атомы - существуют на протяжении всей работы программы. Можно установить значение символьному атому до момента, пока это значение не поменяется. Установка происходит с помощью функций setf и setq. Эти функции имеют два аргумента, первый не вычисляется, второй вычисляется и является значением первого аргумента. Разница между setf и setq - вычисляются оба аргумента или один.
Если в тексте программе используется установленый уникальный атом, то система использует значение установленного ранее атома. Список свойств - динамическая структура, состоящая из списков вида (имя свойства, свойство).

Функция let - является формой, имеет два аргумента (let ((name1 val1) (name2 val2) ... (namen valn)) body) - namen - имя указателя, valn - значение указателя namen, body - выражение, где можно использовать указатели name1 - namen.
Работа let - сначала выделяется память, потом готовятся значения, после чего происходит связывание в произвольном порядке. При этом в name2 нельзя использовать name1, поскольку ещё не произошло связывание, поэтому при вычислении очередного значения namei нельзя ссылаться на значение предыдущих атомов.
let* - позволяет сослаться на значения предыдущих атомов, но работает менее эффективно.

3.3 Функции, реализующие операции со списками

Данные функции можно разбить на две группы - разрушающие структуру и не разрушающие структуру.
Для работы со списком его необходимо создать, получить доступ и модифицировать.
Функции, разрушающие структуру - изменяют структуру списка. Функции не разрушающую структуру - производят какие-то операции без изменения поданного на вход списка.
Функция append - функция объединяет списки. Количество аргументов может быть произвольным. Функция append работает с копией списка -> не разрушается структура списка, поданного на вход.

```
1  (setf lst1 '(a b))
2  (setf lst2 '(c d))
3
4  lst1 -> [|] -> [|] -> nil
5          |       |
6          a       b
7
8  lst1 -> [|] -> [|] -> nil
9          |       |
10         c       d
11
12 (setf lst3 (append lst1 lst2))
13
14 lst3 -> [|] -> [|] -> [|] -> [|] -> nil
15         |       |       |       |
16         a       b       c       d
```

Создаются копии списков, кроме последнего. После чего копии сцепляются последними указателями, последний прицепляется к lst2. В результате можно работать со списками lst1 и lst3, однако lst2 одновременно входит в lst3, что приводит к тому, что его изменении происходит и изменение lst3.
nconc - дубль append, nconc переставляет указатели, в результате чего списки сшиваются без создания дублей
reverse - развернуть с копиями nreverse - развернуть без копий

last - на вход подаётся структура, состоящая из списковых ячеек, возвращает последнюю списковую ячейку.
Список может быть смешанным - числа и символы могут быть в списке. Может быть одноуровневым и многоуровневым. Структурированный/неструктурированный.
Список числовой - только из чисел.
<!> Все стандартные функции работают только с верхним уровнем списка.
(nth N lst) - n-й элемент. (nthcdr N lst) - n-й хвост. (remove elem lst) - удаление элемента, не обрабатывает внутрениие списки.
В стандартных функциях для сравнения использует функцию equal, принимающую на вход только атомы, не списки.
member - проверка присутствия элемента в списке. (member elem lst) - по верхнему уровню ячеек, используется функци eql, не сравнивается список. Пример (member '(a, b) '(c (a b) d)) - в случае подачи, система не обнаружит, поскольку eql не сравнивает списки. Можно повлиять на работу функций с помощью механизма ключевых слов. При создании функции используется набор ключевых параметров, которые могут быть изменены для изменения поведения функции. В данном случае изменение ключевого параметра test позволит включить сравнение списков - (member '(a, b) '(c (a b) d): test #'equal). Результат работы будет ((a b) d).
Для возврата списочных ячеек по элементу - assoc и rassel.
Работа со множествами - работа со множествами, представленными в виде списком. Существуют стандартные функции для работы с множествами.
Ассоциативные таблицы - самый простой вариант реализации - список точечных пар, где в каждой паре точечная пара состоит из ключа и значения. Для работы с ассоциативными таблицами существует несколько стандартных функций.

3.4 Функционалы

Какие типы алгоритмов существуют? - линейный, разветвлённый, циклический.
Для организации повторных вычислений используются либо функционалы, либо рекурсии.
Функции применяющие и отображающие.
Значения глобальных атомов могут влиять на работу функции, поэтму необходимо его заблокировать. # - функциональная блокировка, фиксирует значения глобальных атомов (фиксирует состояние в памяти окружения данной функции). # - function - функция, вызывающая функциональную блокировку.
Функция apply - применяет лямбда-функцию к какому-либо выражению (apply #'fun lst), fun - функция или лямбда-выражение, lst - список аргументов. (funcall #'fun arg1 ... argn) - применение формы, аналогично apply но без списка.
Отображающие фукнционалы - функционалы, позволяющие реализовать многократные или повторные вычисления. Результаты огранизуются в виде списка результатов для каждого вызова функции.

Основные отображающие функционалы: - (mapcar #'fun lst1) - функция, указанная fun, применяется к каждому элементу списка из верхнего уровня ячеек, элементы могут быть как указателями так и списками, поэтому необходимо учитывать это при написании функции. - (maplist #'fun lst2) - функция применяется целиком к списку, к хвосту списка, к хвосту хвоста... пока список не станет пустым. В данной форме функция fun должна быть одноаргументая, в функции maplist - функция fun должна уметь работать со списком.

3.5 Среда

Термин "среда"употребим, когда используется какой-то пакет. При загрузке пакета становится доступен интерфейс среды - набор возможностей, облегчающих программирование.

4 Лекция 4

4.1 Функции работы со списками-

Вызов функции mapcar для функции нескольких аргументов: (mapcar #'fun lst1 lst2 ... lstk). В данном случае mapcar выбирает из каждого списка car аргументы и применяет их к функции. mapcar не контролирует длину списков, поданных на вход. Если списки имеют разную длину, когда заканчиваются элементы по верхнему уровню самого короткого списка.
(maplist #'fun lst1 lst2 ... lstk), функция fun должна работать со списками.
При работе данных функций образуются несколько результатов, которые объединяются в список с помощью list. Существуют дубли, где объединение происходит с помощью функции nconc - mapcan, mapcon.
(find-if #'fun lst) - функционал find-if проходит только по верхнему уровню списковых ячеек, возвращает первый элемент списка, удовлетворяющий данной функции (#'fun = #'predicat). Пример: (find-if #'odd '(2 4 7 5)) → 7. (find-if-not #'predicat lst) - первый элемент, fun от которого возвращает Nil.
(remove-if #'predicat lst)/(remove-if-not #'predicat lst) - удаление элемента с условием.
Если определение предиката написано прямо в функции, то при выполнении программы не происходит множественного перехода по указателю от имени к определению функции, поэтому данный вариант предпочтительнее, так как работает быстрее.
(reduce #'fun lst) - каскадная функция, fun должна иметь не менее двух элементов. Сначала применяет к первым двум элементам, потом к результату и третьему, потом к результату и четвёртому.
(every #'fun lst), (some #'fun lst) также работают каскадным образом.
Примеры применения функционалов:

```
1  (defun consist-of (lst)
2      (if (member (car lst) (cdr lst)) 1 0))
3
4  (defun all-last-element (lst)
5      (if (eql (consist-of lst) 0) (lst (car lst)) ()))
6
7  (defun collection-to-set (lst)
8      (mapcon \#'all-last-element lst))
9
10 (collection-to-set '(i t i g t k s i f k))
11 > (g t s i f k)
```

Смысл примера - порядок результата может зависеть от порядка аргументов в исходном списке.

```
1  (defun dcart (lstx , lsty)
2      (mapcan #'(lambda (x)
```

```
3           (mapcar #'(lambda (x) (list x y)) lsty))
4               lstx))
5
6  (decart '(a b) '(1 2))
7  > ((a 1)(a 2)(b 1)(b 2))
```

Смысл примера - применение лямбда-функций и использование вложенного вызова функционала с фиксацией аргумента x.

5 Лекция 5

5.1 Рекурсия

Рекурсия - повторный вызов некой функции этой функци. Рекурсия - ссылка на некий объект в описании этого объекта. Основные вопросы при создании рекурсии - как войти, как выйти, как передать аргументы.

Классификация рекурсий в Lisp:

— Простая рекурсия - один вызов в теле функции

— Рекурсия первого подярка - когда в теле функции вызов функции производится несколько раз

— Взаимная рекурсия - в теле функции вызывается несколько разных рекурсивных функций

Первые ветки рекурсии реализуются cond - базисный случай. Золотое правило создание рекурсии - сначала проверка, нужно ли выйти из рекурсии и только потом уход на следующий шаг.

Для эффективной реализации рекурсии необходимо заранее выполнить отложенные вычисления и только затем уходить в рекурсию для того, чтобы вернуть только промежуточный результат.

Хвостовая рекурсия - один из способов эффективной организации рекурсии.

lisp, append, cons - какую из этих функций нужно использовать, чтобы реализовать рекурсию эффективно (точно не append, поскольку append делает копии).

"Хорошая"функция - эффективность с точки зрения данных и реализации.

```
1 (defun my-number (el lst)
2     (cond ((null lst) Nil)
3         ((equal el (car lst)) t)
4             (t (my-member el (cdr lst ))))))
5
6 (my-member 'a (b a c))
7 > t
8 (my-member nil ())
9 > nil
```

В последнем вызове вывод не совсем корректный из-за порядка следования проверок cond. Переставив их местами мы можем добиться правильного вывода.

```
1 (defun my-number (el lst)
2     ((equal el (car lst)) t)
3         (cond ((null lst) Nil)
4             (t (my-member el (cdr lst ))))))
5
6 (my-member nil ())
7 > t
```

Реализация собственной функции reverse с помощью append - неэффективный способ. Данная рекурсия нехвостовая, так как результат не вычисляется на входе.

```
(defun my-reverse (lst)
    (cond ((null lst) lst)
        (t (append (my-reverse (cdr lst))
                   (cons (car lst) lst)))))
```

Реализация собственной функции reverse с помощью свой функции - эффективный способ.

```
(defun my-reverse (lst)
    ...)

(defun move-to (lst result)
    (cond ((null lst) result)
    (t (move-to (cdr lst) (cons (car lst) result)))))
```

6 Лекция 6

6.1 Рекурсия

Эффективный вариант реализации рекурсии - при входе не остаётся невыполненных команд, т.е. рекурсивный вызов последний.

Количество веток реализации рекурсии зависит от конкретной задачи.

Может быть несколько выходов из рекурсии и несколько входов в рекурсию.

Общий вид дополняемой рекурсии

```
(defun fn (x)
    (cond (cnd-test end-value)
        (t (cons add_val(fn changed_x)))))
```

Внутри дополняемых рекурсий существует вариант дополняемой рекурсии, имеющей преобразование с условием.

```
(defun fn (x)
    (cond (cnd-test end-value)
        (add-test add-func (fn chaged1_x))
        (t (fn changed2_x)
```

Пример:

```
(defun extract_symbols (lst)
    (cond ((null lst) nil)
    ((symbolp (car lst)
        (cons (car lst)
            (extract-symbols (cdr lst)))))))
```

Множественная рекурсия - может быть дополнительная функция, обрабатывающая два рекурсивных вызова, по голове и хвосту.

Пример:

```
(defun cons_sells (lst)
    (if (atom lst) 0
        (+ (length lst)
            (reduce #'+
                (mapcar #'cons_sells lst)))))
```

Пример:

```
(defun into_one_level (lst rst)
    (cond ((null lst) rst0
        ((atom lst) (cons lst rst))
        (t (into_one_lecel (car lst)
                    (into_one_lecel (cdr lst) rst))))))
```

boundp - связь атома со значением, fboundp - связь атома с функционалом. Свойства - упорядоченный список чётного количества элементов. putprop - назначение свойства. remprop, symb-plist.

Механизм ключевых слов:

&optional - не указанный параметр, если не введён, то заменяется на nil.

&rest Пример:

```
1 (defun f1 (x &optional y) (list x y))
```

7 Лекция 7

7.1 Предпосылки появления Prolog

Процесс совершенствования техники застопорился из-за отсутсвия новой элементной базы. Пролог - реализация логики предикатов в виде языка программирования. При доказательстве теорем не используются данные в привычном смысле слова.
Пролог работает со знаниями.
Основной элемент математический логики - высказывание, которое может быть либо истинно либо ложно.
Задача пролога - автоматизировать процесс доказательства теорем. Для решения данной задачи необходимо было доказаь принцип резолюции. Данный принцип был доказан в 1969 году. Позволяет сделать обоснованный вывод из утверждения.
Пролог поддерживает декларативный способ программирования.
Всё что нужно для функционирования исскуственного интеллекта - обладание знаниями.

7.2 Prolog

Программа на прологе представляет из себя базу знаний. Данная база состоит из аксиом (фактов) и теорем. База знаний фиксируется в разделе, имеющем заголовок clauses - предложениия. Активизация производится в разделе go (цель).
Единственная конструкция пролога - term, либо константа, либо переменная, либо составной терм. Логическая система возвращает либо да либо нет и попутно информация о том, как дойти для нужного результата.
Основная единица - символьный атом, использующегося для обозначения предикатов. С маленькой буквы - константа, с большой буквы - переменная.
Строковые константы - в кавычках. Именованные переменные - переменные использующие имена. Prolog использует анонимные переменные, обозначающиеся как _.
Составные термы используются для того, чтобы зафиксировать, что между какими-то объектами есть связь. F(t1, t2, ..., tn), F - главный функтор, t1-tn - термы.
Пример:
student(ivanov, mgtu). - конкретный студент конкретного вуза
student(X, mgtu). - все студенты конкретного вуза
student(X, Y). - все студенты всех вузов

Все предложения заканчиваются точкой. Количество аргументов в функторе - арность.
student(ivanov) - арность 1, поэтому разные знания.

В момент фиксации утверждение с переменной значение не имеет. Поэтому процесс работы системы заключается в том, чтобы найти решение данного предиката.

Когда система подбирает для переменной значение говорят, что система конкретизирует переменную значением. При этом при подборе система может ошибиться, поэтому система ведёт доказательство путём проб и ошибок. Работа системы - процесс доказательства, работающий на основе фактов. Активация происходит путём задания вопроса.
Синтаксическая форма правила:
A :- B1, B2, ..., Bk.
Факты, содержащие переменные - основные, не содержащие - неосновные. A - заголовок правила. B1-Bk - тело.
student(X, mgtu) :- документы(X, att), ...
Заголовок является фиксацией знания о том, что между аргументами возможна истинная связь (X действительно студент, если...)

Особенный способ работы с переменными:
Вместо того, чтобы сначала задавать значения переменной, задаётся условие и система ищет такие значения переменных, чтобы на вопрос ответить да. При этом сохраняюстя и возвращаются значения для этого ответа.
Переменные нужны для передачи данных во времени и пространстве.
Во времени - переменная получила значение и через какое-то время оно было использовано
В пространство - передача между областями данных (памяти) - перенос в физическом пространстве

Во время фиксации переменная не имеет значения.

8 Лекция 8

База знаний - факты и правила. Факты без переменных - основные, иначе - неосновные. Использование переменных повышает уровень абстракции. Именованные переменные служат для передачи значений во времени и пространстве. Защита лаб - решение объясняется пошагово.
Вопрос задаётся в виде терма, являющегося единственной синтаксической конструкцией языка. Знания всегда находятся в заголовке. Переменные в вопросе позволяют системе подобрать значения и вернуть значения из базы во внешний мир. Анонимные переменные не конкретизируются системой. Пролог основан на методе резолюции - алгоритме унификации. Программа - база знаний и вопрос. В фактах и правилах переменные имён входят с квантовом всеобщности. В вопрос - с квантором существования. Именованная переменная уникальна в пределах предложения.
Правило это заголовок и тело. Заголовок представляется в виде составного тела (либо характеристика объекта, либо отношения между объектами). Является фиксацием знания. Тело - это условие истинности правила.
Составной терм - главный функтор (аргументы). Главный функтор работает как имя отношения.

8.1 Процедуры и декларативные особенности Prolog

Система подбирает значения по сравнению двух составных терм - вопроса и заголовка правила. Данное сравнения происходит с помощью алгоритма унификации.

8.2 Подстановка и примеры термов

Термы без переменных - основные.

Подстановкой называют множество пар вида:
$\{X_i = t_i\}$, где t_i - терм не содержащий переменных (в лучшем случае не содержащий X_i). То есть t_i - значение для X_i.
$\Theta = \{X_1 = t_1, X_2 = t_2, ..., X_n = t_n\}$
$B = A\Theta$ - факт применения подстановки к терму. Применение заключается в замене переменных X_i на соответствующее значение t_i

B называется примером A, если существует Θ, что $B = A\Theta$.
Терм C называется общим примером термов A и B, если существуют такие Θ_1 и Θ_2, что $C = A\Theta_1$, $C = B\Theta_2$

8.3 Логические программы: процедуры

Если знание формируется несколькими предложениями, то это процедура.

Процедура - совокупность правил, заголовки которых имеют одинаковые главные функторы и одинаковые количества аргументов, обозначенными переменными одной природы.
Декларируем наличие связей между объектами.

8.4 Механизм логического вывода

Правила вывода - утверждение о взаимосвязи между допущениями и заключениями которые с в соответствии с правилами логики исчисления предикатов верны всегда.
1. Факты не содержат терм и вопрос основной.
Правило - совпадение
2. Факты основные, а вопрос не основной.
Правило - обобщения папка.
3. Факты содержат терм, а вопрос основной
Правило - конкретизация факта
4. Переменные и там и там
Для ответа система должна построить пример. Сначала конкретизация факта, а затем правилом обобщение.

8.5 Унификация термов

Унификация - операция, которая позволяет формализовать процесс логического вывода. Основной вычислительный шаг, с помощью которого происходит:
1. Двунаправленная передача параметров процедурам
2. Конкретизация
3. Проверка условий

$T_1 = T_2$ - знак равно запускает алгоритм унификации.
Правила унификации:
1. Если T_1 и T_2 константы - если они совпадают.
2. Если T_1 не конкретизированная переменная, а T_2 - константа, или составной терм, не содержащая T_1, то унификация успешна, а T_1 конкретизирует значение T_2.
3. Если T_1 и T_2 не конкретизированные переменные, то их унификация всегда успешна, T_1 и T_2 становятся сцепленными (двума именами одного объекта).
4. Если T_1 и T_2 составные термы, то они успешно унифицируются, если:
1) T_1 и T_2 имеют одинаковый главный функтор
2) T_1 и T_2 имеют равные арности
3) успешно унифицирована каждая пара их соответствующих компонентов

9 Лекция 9

Алгоритм унификации - основной шаг доказательства, система умеет использовать только его. Основное назначение - найти значение переменных в вопросе такое, что на вопрос можно ответить да. Алгоритм унификации это процесс, при этом алгоритм использует области памяти с определённой целью. Информация, использующаяся в процессе унификации разделена.
Урощённо эти области:
1. Рабочее поле
2. Стек для хранения равенств, унификация которых проверяется
3. Результирующая ячейка - область памяти, в которой накапливается подстановка

Назначение алгоритма унификации - проверить, удаётся ли два терма унифицировать.
Также алгоритм унификации использует переменную типа флаг, называемый "неуспех т.е. сигнал о том, произошла унификация или нет. При унификации проверка либо удаётся, либо нет.
0 - успех, 1 - неуспех.
Пусть есть два терма - T_1, T_2.
$T_1 = T_2$ - знак равно указывает на необходимость унификации термов.
Алгоритм:

```
1  Начало
2      Занести в стек  T_1 = T_2
3      Положить        неудача = 0
4      Пока        стек не пуст
5          Считать          из стека в рабочее поле S = T
6          Обработать  если S и T - несовпадающие константы, то неудача = 1, и выход из
               цикла
7                      если S и T - совпадающие константы, то следующий шаг цикла (на ко
                           нец цикла)
8                      если S - переменная, а T - терм, содержащий S в качестве аргумент
                           а, то неудача равна 1 и выход из цикла
9                      если S - переменная, а T - терм, не содержащий S, то для S это во
                           зможное её значение - то отыскать в стеке и результрующей ячей
                           ке все вхождения S и заменить их на T. Добавить в результирующ
                           ую ячейку равенство S = T, переход на конце циклаы
10                     если S и T составные термы, имеющие разные функторы или разную ар
                           ность, то неудача равна 1 и выход из цикла
11                     если S и T составные термы, имеющие одинаковые функторы и арность
                            S = f(S_1, S_2, S_3, ..., S_n) и T = f(T_1, T_2, ..., T_n) то
                            занести в стек равенство S_1 = T_1, ..., S_n = T_n.
12         Очистить рабочее поле
13     Конец цикла
14     Проверка        если неудача 1, унификация невозможна
15                 если неудача 0, унификация успешна, результирующая ячейка содержит по
                       дстановку - эта подстановка называется наиболее общим унификатором
                       .
```

Терм S называется более общим чем T, если T является примером S и S не является примером T.

Терм S называется наиболее общим двух термов T_1 и T_2, если S является более общим по отношению к любому другому их примеру.

Унификатором двух термов называется подстановка, которая будучи применена к двум термам даст одинаковый результат.

Наиболее общим унификатором двух термов называется унификатор соответствующих наиболее общему примеру двух термов.

Теорем:

Если два терма унифицируемы, то существует единственный, с точностью до переименования переменных наиболее общий унификатор, а значит наиболее общий пример.

Поэтому алгоритм унификатор строит наиболее общий пример.

Пример:

есть выражение t(X, p(X, Y)) = t(q(W), p(q(a), b))

результирующая ячейка	рабочее поле	стек
-	-	t(X, p(X, Y)) = t(q(W), p(q(a), b))
-	t(X, p(X, Y)) = t(q(W), p(q(a), b))	X = q(W), p(X,Y) = p(q(a), b)
X = q(W)	X = q(W)	p(X,Y) = p(q(a), b)
X = q(W)	p(q(W),Y) = p(q(a), b)	q(W) = q(b), Y = b
X = q(W)	q(W) = q(a)	W = a, Y = b
X = q(a), W= a	W = a	Y = b
X = q(a), W = a, Y = b	Y = b	-

При одном запуске система определяет, подходит это знание для вывода, или нет.

9.1 Общая схема согласования целей

Решение логической задачи начинается с вопроса G и заканчивается получением 1 и более результатов:

1) Успешного согласования вопроса и программы a в качестве побочного эффекта - подстановка, которая содержит значение переменных, при которых вопрос является примером программы (или нескольких таких решений) 2) неудача

Вычисление с помощью конечной логической программы представляет собой пошаговое преобразование логического вопроса.

На каждом шаге имеется некая коньюнкция целей, выводимость которой следует доказать.

Эта коньюнкция целей называется резольвентой. Успех достигается, когда резольвента пуста.

Преобразование резольвенты выполняется с помощью редукции.

Редукцией цели G с помощью программы P называется замена цели G телом того правила из P, заголовок которого унифицируется с целью G. Такие правила называются сопоставимыми с целью.

Новая резольвента образуется в 2 этапа: 1) В текущей резольвенте выбирается одна из целей и для неё вычисляется редукция. 2) Затем к полученной коньюнкции целей применяется подстановка, полученнная как наиболее общий унификатор цели и заголовка сопоставляемого с ней правила.

Если в результате работы алгоритма унификации для редукции был подобран факт, то резольвента уменьшится на один терм.