Карпов В.Э.

Формальные системы

Определения

Формальная система (ФС) определяется как четверка

$$ФС = \{\Sigma, F, A, P\}, \text{ где}$$

- Σ - конечный алфавит (конечное множество символов, словарь).
- **F** - процедура построения формул (слов) формальной системы (грамматика формул). Можно трактовать как определение того, что такое формула.
- **A** - конечное множество формул, называемых *аксиомами*.
- **P** - конечное множество *правил вывода*, которые позволяют получать из некоторого конечного множества формул другое множество формул:

$$U_1, U_2,\ldots,U_n \rightarrow W_1, W_2,\ldots W_m$$

U_i, W_i – формулы ФС.

Определения

Определение. *Формальное доказательство* – это конечная последовательность формул M_1, M_2, …, M_r таких, что каждое M_i – либо аксиома, либо M_i выводима при помощи одного из правил вывода из предшествующих формул M_j, $j<i$.

Определение. Формула t называется *теоремой*, если существует доказательство, в котором она является последней:

$$M_r \equiv t$$

В частности, при таких определениях любая аксиома является теоремой.

Существует 2 типа правил вывода:

- **Продукции**. Применяются к формулам, рассматриваемым как единое целое:
  $tA \rightarrow tAS$
- **Правила переписывания**. Применяются к любой отдельной части формулы:
  $SS \mid\!\rightarrow S$

Исчисление высказываний

Примером ФС является исчисление высказываний (ИВ). ИВ – это ФС, называемая также логикой высказываний (пропозициональной логикой).

$$\frac{\Phi, \quad \Phi \to \Psi}{\Psi}$$

Алфавит ИВ:

- Пропозициональные буквы p, q, r, s, t, …
- Логические операторы ¬ (не), ⊃ (влечет, следует)
- Скобки '(', ')'.

Построение формул ИВ:

- Любая пропозициональная буква суть формула.
- Если m-формула, то и (m) является формулой.
- Если m-формула, то ¬m является формулой.
- Если m1 и m2 – формулы, то m1 ⊃ m2 – формула.

Аксиомы ИВ:

- (A1): m1 ⊃ (m2 ⊃ m1)
- (A2): (m1 ⊃ (m2 ⊃ m3)) ⊃ ((m1 ⊃ m2) ⊃ (m1 ⊃ m3))
- (A3): (¬m2 ⊃ ¬m1) ⊃ (m1 ⊃ m2)

Правила вывода:

- m1 и (m1 ⊃ m2) |→ m2 (*modus ponens* или правило отделения)

В этой ФС 3 закона логики Аристотеля являются строго доказуемыми:

- Закон тождества: (p ⊃ p)
- Закон исключения третьего: (p∨¬p)
- Закон противоречия: (¬ (p∧¬p))

Введение новых символов:

- ∧ (И): m1∧m2 ↔ ¬(¬m1⊃m2)
- ∨ (ИЛИ): m1∨m2 ↔ ¬(¬m1∧¬m2)
- ≡ (ТОЖДЕСТВО): m1≡m2 ↔ (m1⊃m2) ∧ (m2⊃m1)

Если истинно Φ и истинно, что из Φ следует Ψ, то Ψ также будет истинным.
«Каждый человек смертен. Сократ — человек. Следовательно, Сократ смертен». (modus Barbara, т.к. здесь имеются кванторы всеобщности).

Множество ИВ

Можно построить множество различных систем аксиом ИВ. В том числе, возможно *уменьшение числа аксиом и количества операторов:*

Оператор - т.н. *штрих Шеффера* и одна аксиома.

Штрих Шеффера определяется следующим образом:

$$p \mid q \leftrightarrow (\neg p) \vee (\neg q)$$

Аксиома Дж.Нико, 1916:

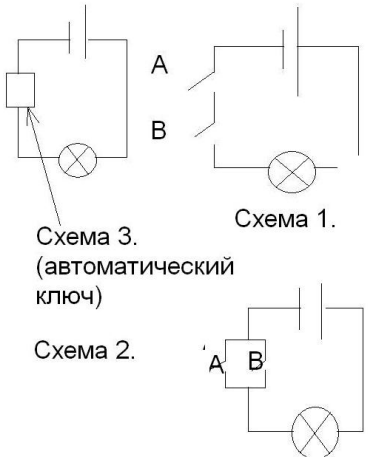
$$(A|(B|C)) \mid \{[D \mid (D \mid D)] \mid [(E|B| ((A|E) \mid (A|E))]\}$$

ИВ и теория релейно-контактных схем

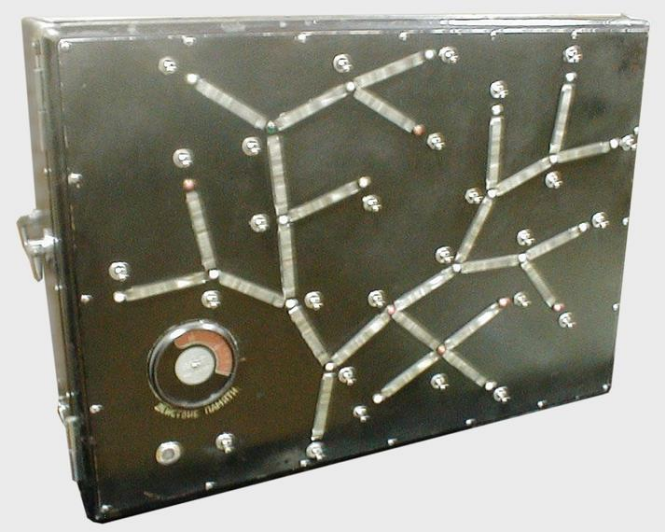
1910 г.: Пауль С. Эренфест (1880-1933): *"...Пусть имеется проект схемы проводов автоматической телефонной станции. Надо определить: 1) будет ли она правильно функционировать при любой комбинации, могущей встретиться в ходе деятельности станции; 2) не содержит ли она излишних усложнений. Каждая такая комбинация является посылкой, каждый маленький коммутатор есть логическое "или-или", воплощенное в эбоните и латуни; все вместе - система чисто качественных... "посылок", ничего не оставляющая желать в отношении сложности и запутанности...правда ли, что, несмотря на существование алгебры логики, своего рода "алгебра распределительных схем" должна считаться утопией?".*

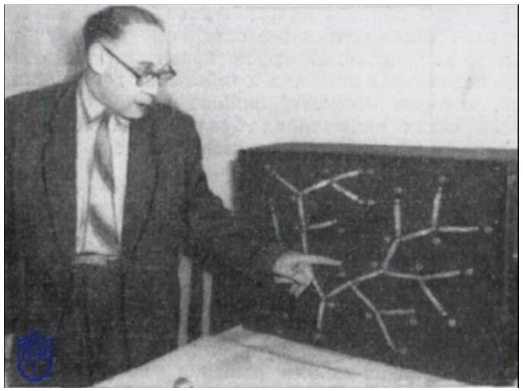
1938 г.: В. И. Шестаков (СССР), К. Шеннон (США) - строгое обоснование возможности использования исчисления высказывании для описания релейно-контактных схем.

1950 г., монография М.А. Гаврилова (1903-1979) "Теория релейно-контактных схем" .

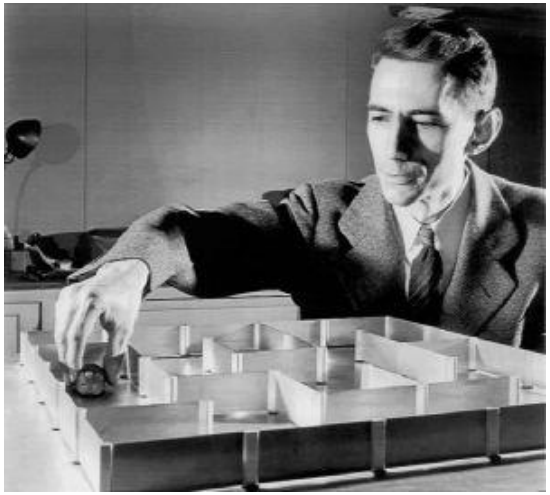


«Мышь в лабиринте»





- 1957, Самообучающиеся автоматы

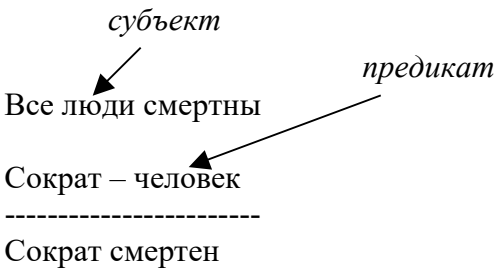


К.Шеннон. "Тесей " ("мышь" Шеннона). Алгоритм Люка-Тремо, **1882**

Язык предикатов первого порядка

Операции с высказываниями, расчлененными на *субъекты* и *предикаты*. Базируется на ИВ. Отсюда следует необходимость понимания внутренней структуры посылок. Появляются кванторы («все», «существует»).

- Алфавит языка предикатов первого порядка включает в себя:
  - разделители
  - константы
  - переменные
  - предикаты
  - функции
  - логические операторы ($\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$)
  - кванторы ($\exists$, $\forall$)



- **Предикат** (логическая функция) – это функция от любого числа аргументов, принимающая значения *Истина* и *Ложь*.
- **Терм** – выражение, включающее константы, переменные и функции.
- *В исчислении предикатов 1-го порядка запрещено квантифицировать символы предикатов и функций.*

К аксиомам логики предикатов добавляются:

- $\forall$ x F(x) $\rightarrow$ F(y)
- F(y) $\rightarrow \exists$x F(x)

Правильно построенные формулы и предложения

Правильно построенная формула (ППФ)

- всякий атом есть ППФ
- если G и H – ППФ, а X – переменная, то
  $(\neg H)$, $(G \wedge H)$, $(G \vee H)$, $(\exists X)G$, $(\forall X)H$ – ППФ.

Предложение – формула, представляющая собой дизъюнкцию литералов.

Удобная и конструктивная форма записи.

Преобразование ППФ в предложения

1. Исключение символов эквивалентности и импликации
   $F \leftrightarrow G \equiv (F \rightarrow G) \wedge (G \rightarrow F)$
   $F \rightarrow G \equiv \neg F \vee G$
2. Уменьшение области действия знаков отрицания. Отрицание должно быть применено не более, чем к одному атому.
   $\neg(F \vee G) \equiv \neg F \wedge \neg G$
3. Разделение переменных. Каждый квантор должен иметь только свою, свойственную ему, переменную.
4. Исключение кванторов существования
   Например, пусть имеем $(\forall X)(\exists Y) P(X,Y)$
   Видимо, Y зависит от X, т.е. $Y=g(X)$ (т.н. *функция Сколема* или сколемовская функция). Тогда можно записать:
   $(\forall X)P(X,g(X))$

Принцип резолюции

1930 г. Эрбран. Метод доказательства теорем в ФС первого порядка.

Идея: чтобы получить некоторое заключение C, исходя из гипотез H1, H2,…Hn, т.е. доказать теорему T:

$$H1 \wedge H2 \wedge Hn \supset C,$$

достаточно доказать <u>противоречивость</u> формулы F:

$$H1 \wedge H2 \wedge Hn \wedge \neg C,$$

в которой отрицание заключения добавлено к исходным гипотезам. Это может оказаться проще прямого вывода.

Соображения тривиальны: доказательство выполнимости множества $G \rightarrow H$ ($\neg G \vee H$) – это опровержение его невыполнимости, т.е. невыполнимости $\neg(\neg G \vee H)$, что эквивалентно $G \wedge \neg H$.

Установление невыполнимости множества предложений осуществляется посредством **принципа резолюций** – специальной процедуры логического вывода новых предложений из множества исходных.

Резольвенты

Пусть имеется два **конкретных** (не содержащих переменных) предложения

$$P1 \vee P2 \vee \ldots \vee Pn \text{ и } \neg P1 \vee Q1 \vee \ldots \vee Qm$$

Из этих двух предложений можно вывести новое предложение, называемое ***резольвентой*** или ***резолюцией***. Резольвентой является дизъюнкция этих предложений с последующим исключением пары P1 и $\neg$P1.

Очевидно, что принцип резолюций покрывает правило вывода *modus ponens*.

Пример. Пусть имеется пара предложений

$$\neg P \vee R \text{ и } \neg R \vee Q \text{ (или } P \rightarrow R,\ R \rightarrow Q)$$

Резольвентой этих предложений является $\neg P \vee Q$ (или $P \rightarrow Q$). Это правило вывода называется *сцеплением*.

Для обобщения этого правила на случай предложений, содержащих переменные, используется специальная процедура, называемая ***унификацией***.

Пусть имеется пара

$$\neg F(X) \vee G(X) \text{ и } F(f(Y))$$

Если первое предложение заменить на

$\neg F(f(Y)) \vee G(f(Y))$, то получим резольвенту G(f(Y)).

Унификация заключается в замене переменных с целью появления дополнительных литералов.

Алгоритм опровержения с помощью резолюций

Предположим, что целью алгоритма резолюций является доказательства $G \rightarrow H$.

Резолюция(G,H)

1. Сформировать C – множество предложений, полученных путем преобразования формул множества G.
2. Добавить к множеству C предложения, полученные из $\neg H$.
3. Пока в C не появится пустое предложение выполнять:

- 3.1. Выбрать 2 различных предложения S1 и S2 из C.
- 3.2. Если они имеют резольвенту R, то добавить эту резольвенту к множеству C.

Конец цикла

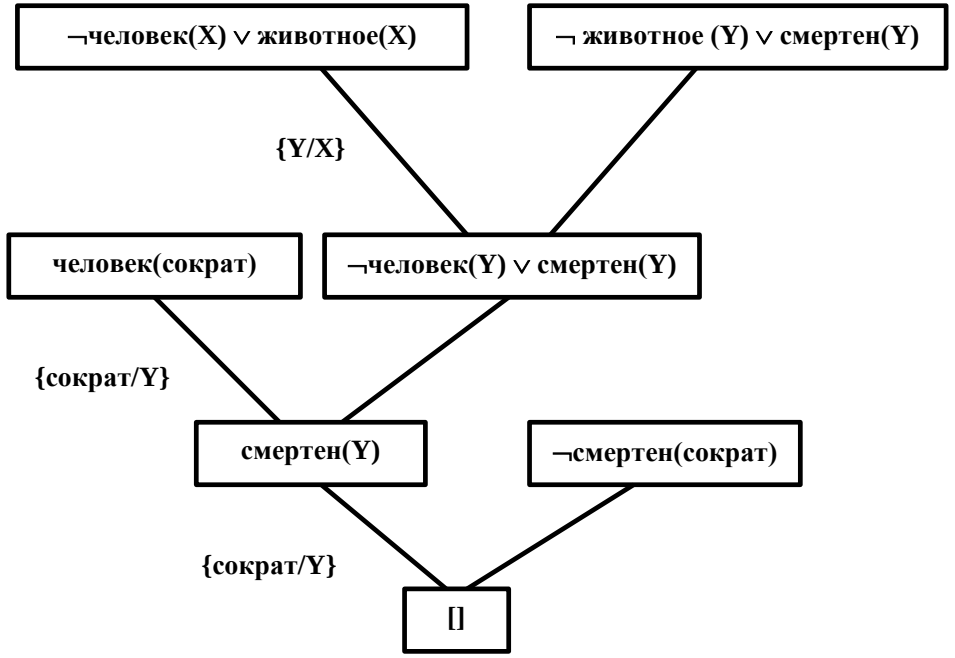
Конец алгоритма.

В этой процедуре требуется предварительное приведение исходных предложений к дизъюнктивной нормальной форме.

Пример 1

Необходимо доказать, что Сократ смертен. Доказательство будем строить на опровержении противоречивости целевого утверждения, т.е. противоречивости *¬смертен(сократ)*.

Предикатная форма	**Дизъюнктивная форма**
Исходные предложения (H)	
∀(X)(человек(X) →животное(X))	¬человек(X) ∨ животное(X)
человек(сократ)	человек(сократ)
∀(X)(животное(X) →смертно(X))	¬ животное (X) ∨ смертно(X)
Требуется доказать (G)	
смертен(сократ)	смертен(сократ)



Пример 2

- *Описание: Любой студент, который сдает экзамен по физике и выигрывает в лотерею – счастлив. Известно, что любой удачливый или старательный студент может сдать все экзамены. Сидоров не относится к числу старательных студентов, но достаточно удачлив. Любой удачливый студент выигрывает в лотерею.*
- *Вопрос: счастлив ли Сидоров?*

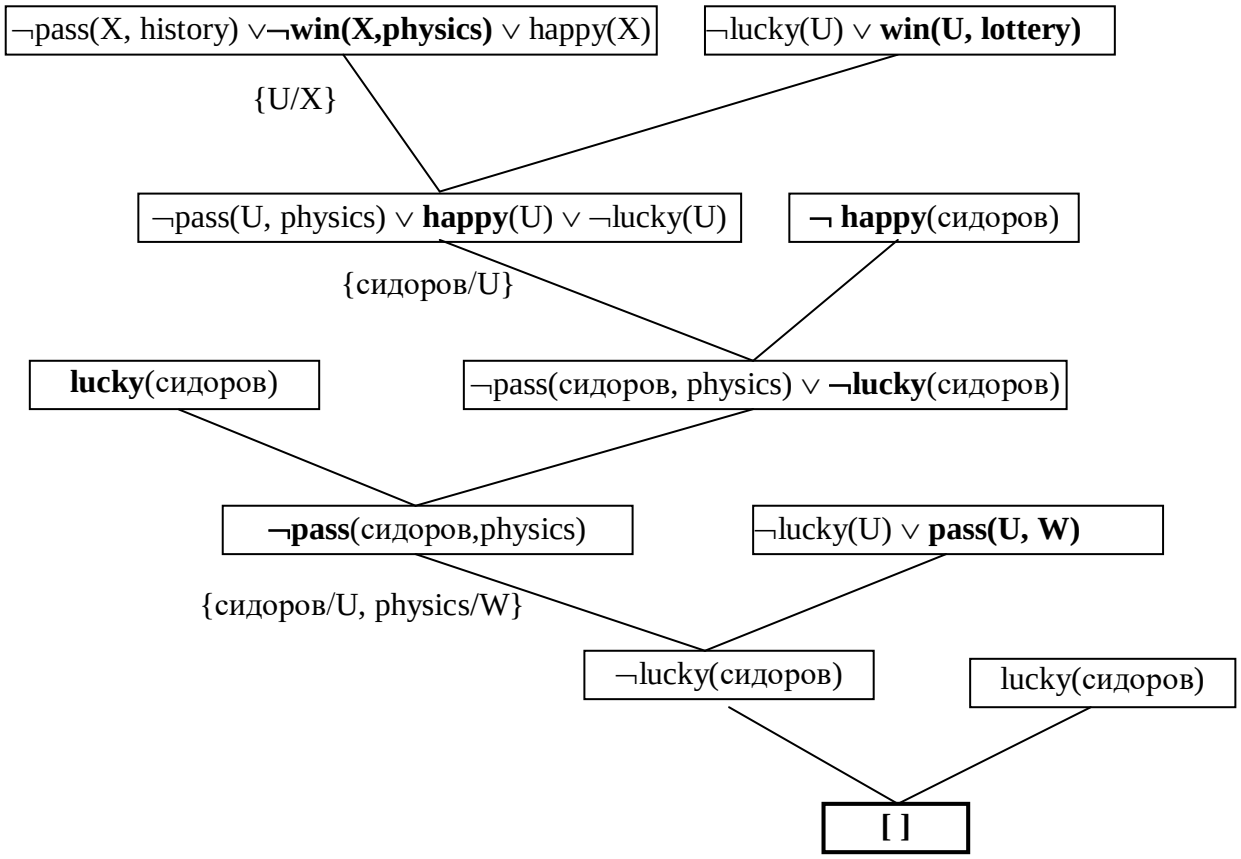
Предикатная и дизъюнктивная формы

Исходное предложение	Высказывания предикатов	Дизъюнктивная форма
Любой студент, который сдает экзамен по физике и выигрывает в лотерею – счастлив.	$\forall X(pass(X, physics) \wedge win(X, lottery) \rightarrow happy(X))$	$\neg pass(X, physics) \vee \neg win(X,lottery) \vee happy(X).$
Известно, что любой удачливый или старательный студент может сдать все экзамены.	$\forall X \forall Y(study(X) \vee lucky(X) \rightarrow pass(X,Y))$	$\neg study(Y) \vee pass(Y,Z).$ $\neg lucky(Y) \vee pass(Y,Z).$
Сидоров не относится к числу старательных студентов, но достаточно удачлив.	$\neg study(сидоров) \wedge lucky(сидоров).$	$\neg study(сидоров).$ $lucky(сидоров).$
Любой удачливый студент выигрывает в лотерею.	$\forall X(lucky(X) \rightarrow win(X, lottery))$	$\neg lucky(U) \vee win(U, lottery).$

Предположим, что мы хотим доказать, что Сидоров счастлив. Для этого добавим к нашим выражениям отрицание заключения ("Сидоров счастлив") в дизъюнктивной форме:

$\neg happy(сидоров).$

Граф опровержения



Граф опровержения разрешения отражает процесс получения противоречия и, следовательно, доказывает, что Сидоров счастлив

Резолюции в Прологе

Пролог основан на логике предикатов первого порядка. В нем используются только предложения Хорна (хорновские дизъюнкты):

$$\neg P_1 \vee \neg P_2 \vee \ldots \vee \neg P_n \vee P_m$$

или

$$P_1 \wedge P_2 \wedge \ldots \wedge P_n \rightarrow P_m$$

При этом:

- резолюция использует первый отрицательный литерал слева направо в центральном предложении
- резолюции выполняются в глубину, т.е. полученная резольвента тут же участвует в следующей резолюции.

Программа доказательства теорем на основе принципа резолюций

Описание исходных предложений:

$$(\neg a \vee b), (\neg b \vee c), a , \neg c$$

```
sent([t(0,"a"),t(1,"b")]). -- (¬a ∨ b)
sent([t(0,"b"),t(1,"c")]). -- (¬b ∨ c)
sent([t(1,"a")]).          -- a
sent([t(0,"c")]).          -- ¬c
```

Здесь 0 - отрицание, 1 - прямая форма

Шаг резолюции состоит из следующих действий:

- Берутся 2 предложения в ДНФ D1 и D2.
- Ищутся дизъюнкты P в D1 и $\neg$P в D2. Если таковые найдены, то формируется дизъюнкция этих предложений с исключение P и $\neg$P.
- Полученная резольвента добавляется в базу (множество предложений).

Текст программы. Начало

```
1. % Программа доказательства теорем на основе принципа резолюций
2. Goal :-
3.   % Загружаем предложения
4.   retractall(_),
5.   consult("resol.db"),
6.   start.

7.   % Вспомогательные предикаты
8.   a2([], L, L).
9.     a2([Head | L1], L2, [Head | L3]) :- a2(L1, L2, L3).
10.  eget([H|T],H).
11.  eget([_|T],R) :- eget(T,R).
12.  remove(L, El, Res) :-
13.    a2(L1,[El|L2],L),
14.    a2(L1,L2,Res).

15.% Запуск программы
16.  start:-
17.    sent(C1),
18.    sent(C2),
19.    resolstep(C1, C2, C1vC2),
20.    not(end(C1vC2)),
21.    % Добавляем резольвенту
22.    asserta(sent(C1vC2)), !,
23.    start.
```

Программа. Продолжение

```
1.  % Шаг резолюции. На входе - предложения C1 и C2. Далее
2.    % из них выделяются дизъюнкты E1 и E2. Дизъюнкты
3.    % сопоставляются и, в случае обнаружения соответствия
4.    % вида (~P и P), формируется резольвента CAvCB. Результат заносится в БД.
5.    resolstep(C1, C2, CAvCB):-
6.      % извлекаем из C1 и C2 элементы E1 и E2
7.      eget(C1, E1), eget(C2, E2),
8.      write("   E1=",E1, ", E2=",E2,"\n"),
9.      % далее сопоставляем подвыражения
10.     compare(E1, E2),
11.     % удаляем подвыражение E1 из C1 и E2 из C2
12.     remove(C1, E1, CA),
13.     remove(C2, E2, CB),
14.     % склеиваем списки, полученные после удаления из них P и ~P
15.     a2(CA, CB, CAvCB), !.
16.   % Проверка на P и ~P - поиск резольвенты
17.   compare(t(N1,X1), t(N2,X2)) :- N1+N2 = 1, unify(X1, X2).
18.   compare(t2(N1,NAME,ARG1), t2(N2,NAME,ARG2)) :- N1+N2 = 1, unify(ARG1, ARG2).
19.   compare(t3(N1,NAME,ARG11,ARG12), t3(N2,NAME,ARG21, ARG22)) :-
20.     N1+N2 = 1,
21.     unify(ARG11, ARG21),
22.     unify(ARG12, ARG22).
23.   % Если список пуст, значит есть противоречие
24.   end([]):- write("\n\nОбнаружено противоречие. Теорема доказана.\n"), exit(0).
25.   % Проверка на унифицируемость
26.   unify(A, A) :- !.
27.   unify(A, B) :- isvar(A), isvar(B), !.
28.   unify(A, B) :- isvar(A), isconst(B), !.
29.   unify(A, B) :- isconst(A), isvar(B), !.
30.
31.   isvar(S) :- frontchar(S,Ch,_), Ch>='A', Ch<='Z', !.
32.   isconst(S) :- frontchar(S,Ch,_), Ch>='a',  Ch<='z', !.
```

Пример с Сидоровым

Исходные данные (файл resol.db) :

1. sent([t2(0,"lucky","U"),t3(1,"win","U","lottery")]).
   $\neg$lucky(U) $\vee$ win(U, lottery)

2. sent([t2(0,"lucky","Y"),t3(1,"pass","Y","Z")]).
   $\neg$lucky(Y) $\vee$ pass(Y,Z).

3. sent([t2(0,"study","Y"),t3(1,"pass","Y","Z")]).
   $\neg$study(Y) $\vee$ pass(Y,Z).

4. sent([t2(0,"study","sidorov")]).
   $\neg$study(сидоров)

5. sent([t3(0,"pass","X",“physics"),t3(0,"win","X","lottery"),t2(1,"happy","X")]). --
   $\neg$pass(X, physics) $\vee$ $\neg$win(X,lottery) $\vee$ happy(X).

6. sent([t2(1,"lucky","sidorov")]).
   lucky(сидоров).
7. sent([t2(0,"happy","sidorov")]).
   $\neg$happy(сидоров)

Алгоритм унификации

1966, Ж.Питра и (независимо от него) Дж.Робинсон на основе работ Эрбрана создают один из основных алгоритмов ИИ – **алгоритм унификации**.

Основной объект - **терм**.

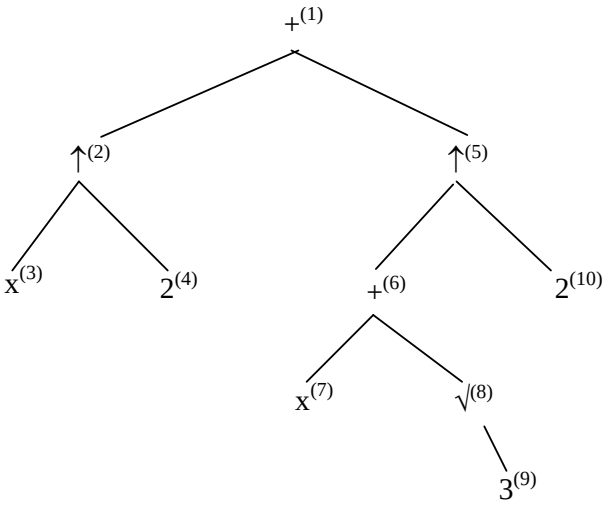
Терм – n-арный объект вместе с n термами.

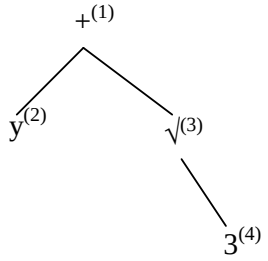
Должны существовать априори исходные термы, являющиеся объектами.

Префиксная форма

$E_{инф} = \log\left(y + \sqrt{y^2 - b / \sin x}\right)$ $E_{преф}$=log + y √ - ↑ y 2 / b sin x

$E_{инф} = x^2+(x+\sqrt{3})^2$ $E_{инф} = y+\sqrt{3}$





1	2	3	4	5	6	7	8	9	10
+	↑	x	2	↑	+	x	√	3	2

1	2	3	4
+	y	√	3

Основная теорема префиксной формы записи

Последовательность символов S является правильно построенной префиксной формой, если и только если:

1. ранг(S)=-1
2. ранг(последовательности слева от S)≥0

Причем считается, что:

ранг(оператора) = вес(оператора)-1,

ранг(пустой последовательности) = 0,

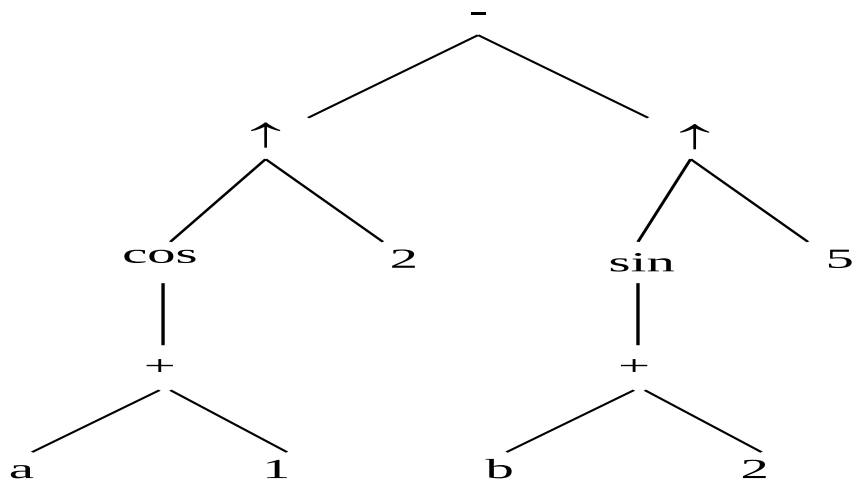
ранг(n-го символа) = n-1,

ранг(переменной) = ранг(константы) = -1,

ранг(S1 соединенной с S2) = ранг(S1)+ранг(S2).

Пример

Пример 7. $E_{инф} = \cos^2(a+1)-\sin^5(b+2)$

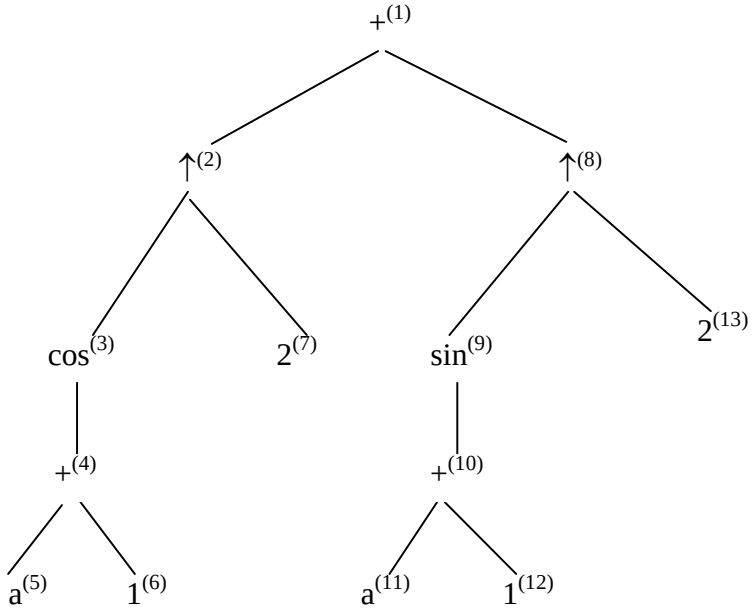


№	1	2	3	4	5	6	7	8	9	10	11	12	13
Симв	-	↑	cos	+	a	1	2	↑	sin	+	b	2	5
Ранг	1	1	0	1	-1	-1	-1	1	0	1	-1	-1	-1

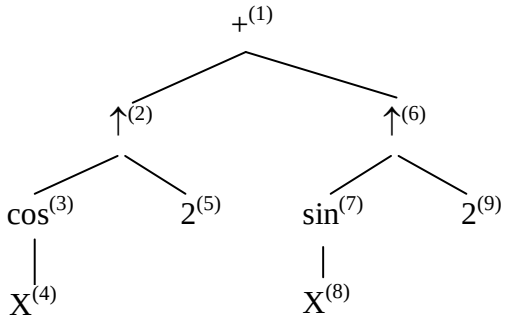
Суммарный ранг равен -1. Для нахождения **границы терма**, образованного, например, оператором cos, начнем суммировать ранги, начиная с этого оператора. Мы остановимся в позиции № 6. Таким образом, граница терма - с 3 по 6 позиции (cos + a 1).

Идея алгоритма

$E_{инф} = \cos^2(a+1)+\sin^2(a+1)$



$T = \cos^2(X)+\sin^2(X)\text{->}1$



$\Rightarrow$ **1**

1	2	3	4	5	6	7	8	9	10	11	12	13
+	↑	cos	+	a	1	2	↑	sin	+	a	1	2

Программа унификации

```
1. #!/usr/bin/env python
2. # coding: utf-8
3. #  Алгоритм унификации
4. from __future__ import print_function

5. # Словарь переменных
6. VarList = {}

7. def AddVar(v, t):
8.     for k in VarList.keys():
9.       if k == v:
10.        if VarList[k] == t: return True
11.        else: return False
12.    VarList[v] = t
13.    return True

14.def replace(S, n, v): return S[:n] + v + S[n+1:]

15. def is_operator(c): return c in['+','-','*','/','\\','~','^','&','|','=']

16.def is_constant(c): return (ord(c) in range(ord('a'), ord('z'))) or \
17.                           (ord(c) in range(ord('0'), ord('9')))

18.def is_var(c): return (ord(c) in range(ord('A'), ord('Z')))
```

Программа. Продолжение

```
1.  #  v - свободная переменная,   t - терм,
2.  #  h1 - индекс в выражении E1,   h2 - индекс в выражении E2
3.  def SUBST(E1, E2, v, t, h1, h2):
4.      # Просмотр терма t на предмет поиска недопустимых в нем переменных
5.      for i in range(len(t)):
6.        if t[i]==v: return False, None, None
7.      # Просмотр E1
8.      for i in range(h1, len(E1)):
9.          if E1[i] == v:
10.           E1 = replace(E1, i, t) # замена символа термом
11.         if not AddVar(v, t): return False, None, None
12.     # Просмотр E2
13.     for i in range(h2, len(E2)):
14.         if E2[i] == v:
15.           E2 = replace(E2, i, t) # замена символа термом
16.         if not AddVar(v, t): return False, None, None
17.     return True, E1, E2

18. def getterm(E, n):
19.     t = ''
20.     cnt = 1
21.     i = n
22.     eoj = False
23.     while not eoj:
24.         t = t + E[i]
25.         if is_operator(E[i]): cnt += 1
26.         else: cnt -= 1
27.         i += 1
28.         eoj = (i>=len(E)) or (cnt==0)
29.     return t
```

Программа. Продолжение

```
1.  #  E1, E2 - унифицируемые выражения
2.  def Unificate(E1, E2):
3.      global VarList
4.      h1 = 0 # начальное выражение - символ из E1
5.      h2 = 0 # начальное_выражение - символ из E2
6.      # Просмотреть параллельно Expr1 и Expr2
7.      VarList = {}
8.      while h2<len(E2):
9.          if E1[h1] != E2[h2]:
10.             # Константы не совпали
11.             if is_constant(E1[h1]) and is_constant(E2[h2]): return False
12.             if is_operator(E1[h1]): # оператор
13.                 if is_operator(E2[h2]): return False # Не совпали операторы
14.                 if is_var(E2[h2]): # E2[h2] - переменная
15.                     # Пробуем заменить оператор на терм. t - терм-дебютант из E1
16.                     t = getterm(E1,h1)
17.                     res, E1, E2 = SUBST(E1, E2, E2[h2], t, h1, h2)
18.                     if not res: return False
19.             else:                       # E1(h1) - переменная
20.                 if is_var(E2[h2]):    # E2[h2] - переменная
21.                     # Пробуем заменить переменную на другую переменную
22.                     res, E1, E2 = SUBST(E1, E2, E2[h2], E1[h1], h1, h2)
23.                     if not res: return False
24.                 else:
25.                     # Пробуем заменить переменную на терм. t - терм-дебютант из E2
26.                     t = getterm(E2,h2)
27.                     res, E1, E2 = SUBST(E1, E2, E1[h1], t, h1, h2)
28.                     if not res: return False
29.         h1 += 1
30.         h2 += 1
31.     return (E1==E2)
```

Программа. Продолжение

```
1.    # T - теорема T=(H,C)
2.    # H(гипотеза) -> C(заключение)
3.    TT = [{'H':'-XX','C':'0'},
4.          {'H':'+0X','C':'X'},
5.          {'H':'+X0','C':'X'},
6.          {'H':'*X1','C':'X'},
7.          {'H':'*1X','C':'X'},
8.          {'H':'/X1','C':'X'}]

9.    def ApplyTheorem(Expr):
10.       for i in range(len(Expr)):
11.           if is_operator(Expr[i]):
12.               t = getterm(Expr,i)
13.               # Проходим по всему списку теорем
14.               for h in TT:
15.                   if Unificate(t, h['H']) :
16.                       Cons = h['C']
17.                       # Подставляем значения переменных
18.                       for k in VarList.keys():
19.                           Cons = Cons.replace(k,VarList[k])
20.                       Expr = Expr[:i] + Cons + Expr[i+len(t):]
21.                       print('  "{}" with ["{}" -> "{}"] => Expr =
      "{}"'.format(t, h['H'], h['C'], Expr))
22.                       return True, Expr
23.       return False, Expr
```

Программа. Окончание

```
1.   def EvalExpr(Expr):
2.       res = True
3.       while res:
4.           res, Expr = ApplyTheorem(Expr)
5.       return Expr

6.   ExprList = ['-aa',            # -> 0
7.               '*1e',            # -> e
8.               '+0e',            # -> e
9.               '+-+-x0x*y1/y1', # -> +-+-x0xyy
10.              '-+x0x',          # -> 0
11.              '-+/+ab10*+ab1', # -> 0
12.              '-+*ab*abc',      # -> False: -+*ab*abc
13.              '+-*ab*ab*1e',    # -> e
14.              '-+*x10*x1',      # -> 0
15.              '-+/+ab10*+ab1', # -> 0
16.              '+-+yy*y1/y1‘]    # -> +-+yyyy

17.  for e in ExprList:
18.      print('Expr =', e)
19.      res = EvalExpr(e)
20.      print('Result: ', res)
```

Пример работы программы

```
1.    Expr = -aa
2.      "-aa" with ["-XX" -> "0"] => Expr = "0"
3.    Result:  0
4.    ----------------------
5.    Expr = *1e
6.      "*1e" with ["*1X" -> "X"] => Expr = "e"
7.    Result:  e
8.    ----------------------
9.    Expr = +0e
10.     "+0e" with ["+0X" -> "X"] => Expr = "e"
11.   Result:  e
12.   ----------------------
13.   Expr = +-+-x0x*y1/y1
14.     "*y1" with ["*X1" -> "X"] => Expr = "+-+-x0xy/y1"
15.     "/y1" with ["/X1" -> "X"] => Expr = "+-+-x0xyy"
16.   Result:  +-+-x0xyy
17.   ----------------------
18.   Expr = -+x0x
19.     "+x0" with ["+X0" -> "X"] => Expr = "-xx"
20.     "-xx" with ["-XX" -> "0"] => Expr = "0"
21.   Result:  0
22.   ----------------------
23.   Expr = -+/+ab10*+ab1
24.     "+/+ab10" with ["+X0" -> "X"] => Expr = "-/+ab1*+ab1"
25.     "/+ab1" with ["/X1" -> "X"] => Expr = "-+ab*+ab1"
26.     "*+ab1" with ["*X1" -> "X"] => Expr = "-+ab+ab"
27.     "-+ab+ab" with ["-XX" -> "0"] => Expr = "0"
28.   Result:  0
```

Пример работы программы

```
1.    Expr = -+*ab*abc
2.    Result:  -+*ab*abc
3.    ----------------------
4.    Expr = +-*ab*ab*1e
5.      "-*ab*ab" with ["-XX" -> "0"] => Expr = "+0*1e"
6.      "+0*1e" with ["+0X" -> "X"] => Expr = "*1e"
7.      "*1e" with ["*1X" -> "X"] => Expr = "e"
8.    Result:  e
9.    ----------------------
10.   Expr = -+*x10*x1
11.     "+*x10" with ["+X0" -> "X"] => Expr = "-*x1*x1"
12.     "-*x1*x1" with ["-XX" -> "0"] => Expr = "0"
13.   Result:  0
14.   ----------------------
15.   Expr = -+/+ab10*+ab1
16.     "+/+ab10" with ["+X0" -> "X"] => Expr = "-/+ab1*+ab1"
17.     "/+ab1" with ["/X1" -> "X"] => Expr = "-+ab*+ab1"
18.     "*+ab1" with ["*X1" -> "X"] => Expr = "-+ab+ab"
19.     "-+ab+ab" with ["-XX" -> "0"] => Expr = "0"
20.   Result:  0
21.   ----------------------
22.   Expr = +-+yy*y1/y1
23.     "*y1" with ["*X1" -> "X"] => Expr = "+-+yyy/y1"
24.     "/y1" with ["/X1" -> "X"] => Expr = "+-+yyyy"
25.   Result:  +-+yyyy
```

Системы автоматического доказательства теорем

Отличие систем автоматического доказательства теорем (или системы автоматизированного формирования рассуждений) от языков логического программирования:

- обычно поддерживают полную логику первого порядка;
- синтаксическая форма, выбранная для высказываний, не влияет на результаты;
- управляющая информация обычно хранится отдельно от БЗ, а не входит в состав самого представления знаний
- большинство исследований в области САД посвящено поиску **стратегий управления**, которые приводят к общему повышению эффективности, а не только к увеличению быстродействия.

Система Otter

Программа автоматического доказательства теорем Otter (Organized Techniques for Theorem-proving and Effective Research).

Подготавливая любую задачу для программы Otter, пользователь должен разделить знания на четыре части:

1. Множество выражений, известное как множество поддержки, в котором определяются **важные факты о данной задаче**. На каждом этапе резолюции операция резолюции применяется к одному из элементов множества поддержки и к другой аксиоме, поэтому поиск сосредоточивается на множестве поддержки.
2. Множество полезных аксиом (usable axiom), которое выходит за пределы множества поддержки. Эти аксиомы предоставляют фоновые знания о проблемной области.
3. Множество уравнений, известных как правила перезаписи (rewrites), или демодуляторы (demodulators). Демодуляторы представляют собой уравнения. Например:
   x+0=x (любой терм в форме x+0 должен быть заменен термом x).
4. Множество параметров и выражений, которое определяет стратегию управления. В частности - задание эвристической функции для управления поиском и функцию фильтрации для устранения некоторых подцелей как не представляющих интереса.

Как работает Otter

1. Принцип постоянного применения правила резолюции к одному из элементов множества поддержки и к одной из полезных аксиом.
2. В отличие от системы Prolog, в этой программе используется определенная форма поиска по первому наилучшему совпадению. Ее эвристическая функция измеряет "вес" каждого выражения с учетом того, что наиболее предпочтительными являются выражения с наименьшими весами. Единичные выражения оцениваются как имеющие наименьший вес.
3. На каждом этапе программа Otter перемещает выражение "с наименьшим весом" из множества поддержки в список полезных аксиом и добавляет в множество поддержки некоторые непосредственные следствия применения операции резолюции к выражению с наименьшим весом и к элементам списка полезных аксиом.
4. Программа Otter останавливается, если обнаруживает противоречие или если возникает такая ситуация, что в множестве поддержки не остается больше выражений.

Алгоритм

procedure Otter(sos, usable)
    inputs:
        sos -- множество поддержки – выражения, определяющие
                -- решаемую задачу (глобальная переменная)
        usable -- множество фоновых значений, которые потенциально
                -- могут быть релевантными для данной задачи
repeat
    clause ← элемент множества sos с наименьшим весом;
    переместить выражение clause из множества sos в множество usable;
    Process(Infer(clause, usable), sos)
until sos=[] **or** <обнаружится опровержение>

function Infer(clause, usable) **returns** множество выражений clauses
    применить правило резолюции к выражению clause и каждому элементу множества usable;
    возвратить полученное множество clauses после применения функции Filter

procedure Process(clauses, sos)
    for each clause **in** clauses **do**
        clause ← Simplify(clause)
        выполнить слияние идентичных литералов;
        отбросить выражение clause, если оно представляет собой тавтологию
        sos ← [clause | sos]
        if clause не имеет литералов, то обнаружено опровержение
        if clause имеет один литерал, то искать единичное опровержение

Standard usage Otter syntax

$\forall$xP(x)	all x P(x)
$\forall$xy $\exists$z (P(x, y, z) $\vee$ Q(x, z))	all x y exists z (P(x,y,z) | Q(x,z))
$\forall$x (P(x) $\wedge$ Q(x) $\wedge$ R(x) $\rightarrow$ S(x))	all x (P(x) & Q(x) & R(x) -> S(x))

Predeclared Functors

- op(800, xfx, ->). op(700, xfx, @<).
- op(800, xfx, <->). op(700, xfx, @>).
- op(790, xfy, |). op(700, xfx, @<=).
- op(780, xfy, &). op(700, xfx, @>=).
- op(700, xfx, =). op(500, xfy, +).
- op(700, xfx, !=). op(500, xfx, -).
- op(700, xfx, <). op(500, fx, +).
- op(700, xfx, >). op(500, fx, -).
- op(700, xfx, <=).
- op(700, xfx, >=). op(400, xfy, *).
- op(700, xfx, ==). op(400, xfx, /).
- op(700, xfx, =/=). op(300, xfx, mod).

```
list(usable).
        x = x. % reflexivity
        f(e,x) = x. % left identity
        f(g(x),x) = e. % left inverse
        f(f(x,y),z) = f(x,f(y,z)). % associativity
        f(z,x) != f(z,y) | x = y. % left
        cancellation
        f(x,z) != f(y,z) | x = y. % right
        cancellation
end_of_list.
```

Расширение системы Prolog

PTTP (Prolog Technology Theorem Prover).

1. В процедуру унификации снова вводится **проверка вхождения** для того, чтобы эта процедура стала непротиворечивой.
2. Поиск в глубину заменяется поиском с **итеративным углублением**. Это позволяет добиться того, чтобы стратегия поиска стала полной, а увеличение продолжительности поиска измерялось лишь постоянной зависимостью от времени.
3. Разрешается применение **отрицаемых литералов** (таких как $\neg P(x)$). В этой реализации имеется две отдельные процедуры; в одной из них предпринимается попытка доказать P, а в другой — доказать $\neg P$
4. Выражение с n атомами хранится в виде **n различных правил**. Например, при наличии в базе знаний выражения $A \Leftarrow B \wedge C$ должно быть также предусмотрено хранение в ней этого выражения, представленного как $\neg B \Leftarrow C \wedge \neg A$ и как $\neg C \Leftarrow B \wedge \neg A$.
5. Логический вывод сделан **полным** (даже для нехорновских выражений) путем добавления правила резолюции с линейным входным выражением: если текущая цель унифицируется с отрицанием одной из целей в стеке, то данная цель может рассматриваться как решенная (один из способов рассуждения от противного).
   Если первоначальной целью было высказывание P и эта цель свелась в результате вывода к цели $\neg P$, то установлено, что $\neg P \Rightarrow P$. А это выражение логически эквивалентно P.