Московский авиационный институт
(государственный ТЕХнический университет)

Факультет прикладной математики

Кафедра вычислительной математики и программирования

Подготовка к экзамену по курсу «Логическое программирование»

Студент: И. К. Никитин
Преподаватели: Д. В. Сошников
М. А. Левинская

Москва, 2009

Содержание

I. Вопрос 1.1

I Основы логического программирования

1 Парадигмы

Различные парадигмы программирования и подходы к определению вычислимости. Декларативные языки программирования как альтернатива императивным. Логическое программирование как алгоритмическая модель, альтернативная модели Тьюринга — фон Неймана

Парадигма	Принцип	+	−
Императивная	**Модель:** Машина фон Неймана основанная на Тьюринге. Программы представляют из себя набор примитивных инструкций, комбинируемых определённым образом. Примитивные операторы 1. Присваивание 2. Ввод-вывод Правила комбинирования 1. Композиция 2. Условная композиция 3. Циклическая композиция	1. естественное соответствие архитектуре	1. непрозрачная семантика 2. присваивание 3. контекст (область видимости) 4. постановка задачи

Логическая	**Модель:** Математическая логика. Логика предикатов первого порядка. Она достаточно формальна и позволяет формулировать свойства.	Программы представляют собой строго формализованную постановку задачи. Естественный способ описания. Выполнение инициализируется запросом на нахождение значений переменных, удовлетворяющих какому-нибудь предикаты или формуле.	реализация: перевод описания на декларативном языке в классический императивный алгоритм.
Функциональная	**Модель:** λ –исчисление. Процедуры манипулируют описание других процедур.	формализм более высокого порядка чем логическое программирование	реализация: перевод описания на декларативном языке в классический императивный алгоритм.
Ситуационная	**Модель:** Нормальные алгоритмы Маркова. Программа представляет собой набор подстановок применяемых в соответствии с определёнными правилами ко входной строке для получения результата.		

Стиль программирования, при котором решение задачи или алгоритм решения генерируется компьютером автоматически, а для решения необходимо только формальное описание, называют **декларативным**

Эквивалентность моделей Множество потенциальных задач шире, чем множество алгоритмов. $\exists$ невычислимые функции, которые невозможно вычислить при помощи алгоритмов в известных алгоритмических моделях.

Теоремма 1. *Множества вычислимых функций в алгоритмических моделях:*

I. Вопрос 1.3

- *машина Тьюринга*
- *исчисление предикатов первого порядка*
- *Нормальные алгоритмы Маркова*
- *рекурсивные функции Черча*

совпадают.

Док-во. Доказывается через эмуляцию. □

Определение 1. *Эффективно вычислимая функция – вычислимая при помощи наформального алгоритма как последовательность дискретных элементарных шагов, детерминированная и конечная.*

Теоремма 2 (Теорема Тьюринга-Чёрча)**.** *Множество функций вычислимых в соответствующей алгоритмической модели. совпадает с множеством эффективно вычислимых функций.*

I. Вопрос 2.1

2 Логика предикатов 1-го порядка

Логика предикатов 1-го порядка. Семантика логики предикатов. Понятия выполнимости и общезначимости. Понятие логического следствия $\models$.

Определение 2. *Логика высказываний (или пропозициональная логика) — это формальная теория, основным объектом которой служит понятие* ***логического высказывания****. С точки зрения выразительности, её можно охарактеризовать как классическую логику нулевого порядка. Логика высказываний является простейшей логикой, максимально близкой к человеческой логике неформальных рассуждений и известна ещё со времён античности.*

Определение 3. *Логика первого порядка (исчисление предикатов) – формальное исчисление, допускающее высказывания относительно переменных, фиксированных функций, и предикатов. Расширяет логику высказываний. В свою очередь является частным случаем логики высшего порядка.*

Логика предикатов первого порядка:

1. Предикаты (высказывание зависящее от параметров)
2. Связанные переменные и кванторы $(\forall, \exists)$
3. Функциональные символы $f(x_1, \ldots x_n)$ как аргументы предикатов.

I. Вопрос 2.2

2.1 Семантика логики предикатов

Предполагают что $\exists$ область интерпретации D. Формулы:

- **Замкнутые** (не содержит вхождений свободных переменных)
  - Придать значения из области интерпретации всем предметным константам a_i:
    $$\aleph_a : \{a_i\} \to D$$
  - Придать смысл всем функциональным символам
    $$\aleph_f : \{f_i^{(n)}\} \to (D^n \to D)$$
  - Придать значения всем предикатным символам
    $$\aleph_p : \{P_i^{(n)}\} \to (D^n \to \{\mathbf{T}, \mathbf{F}\})$$

  Т.о. интерпретация $\aleph = \langle D, \aleph_a, \aleph_f, \aleph_p \rangle$
- **Не замкнутые** (зависит от некоторого набора переменных $\{ x_1, \dots x_n\}$)
  - Однозначная интерпретация невозможна
  - Вводится понятие оценки на наборе $\{a_1, \dots a_n\}$

Определение 4. *Формула F называются **общезначимой** $\models F$ если она истина во всех интерпретациях.*

Определение 5. *Формула F называются **выполнимой** если она истина хотя бы в одной интерпретации.*

Определение 6. *Формула B **следует** из формулы A*

$$A \models B$$

если в любой интерпретации для которой истинно A оказывается истинным и B

Можно распространить на произвольное количество высказываний

$$A_1, \dots A_n \models B \Leftrightarrow A_1 \wedge \dots \wedge A_n \models B$$

Теоремма 3.

$$A \models B \Leftrightarrow \models A \supset B$$

I. Вопрос 3.1

3 Исчисление предикатов 1-го порядка

Исчисление предикатов 1-го порядка. Понятие о формальной аксиоматической системе. Выводимость ($\vdash$) в формальной аксиоматической системе. Правило вывода *modus ponens*.

Определение 7. *Формальная аксиоматическая система*

1. ***Алфавит***
$$A = \{A_0, \ldots, A_n\}$$
некоторое конечное не пустое множество символов

2. ***Множество правильно построенных формул***
$$\Phi \subseteq A^*$$
задаётся грамматикой

3. ***Множество аксиом***
$$\Gamma_0 \subseteq \Phi$$

4. ***Правил вывода*** *позволяют получить из нескольких правильно построенных формул заключения*

   Определение 8. ***Вывод формулы*** f ***из множества формул*** Γ — *последовательность* $\{f_0, \ldots, f_n = f\}$ *где* f_i *получается применением правил вывода к некоторым формулам из* $\Gamma \cup \{f_0, \ldots, f_n = f\}$

   Определение 9. *В этом случае говорят, что* ***формула*** f ***выводимая из множества*** Γ
$$\Gamma \vdash f$$

   Определение 10. ***Формула*** f ***выводимая в формальной аксиоматической системе***
$$\vdash f$$
   если $\Gamma_0 \vdash f$

I. Вопрос 3.2

Определение 11. *Исчисление предикатов – формальная аксиоматическая система с синитаксисом логики предикатов*

1. **Алфавит**

$$A =$$

$$\{\exists, \forall, (,), \vee, \wedge, \neg, \supset\}$$

$$\cup$$

Символы предикатов

$$\cup$$

Символы термов

Определение 12. *Терм:*

(a) *Предметная константа* $a_i \in M$ *– Уникальный объект предметной области*

(b) *Переменная* x_i

(c) *Функциональный терм* $f(t_1 \dots t_n)$*, арности* n*;*
t_i *– термы*

2. **Множество правильно построенных формул**

- Атомарная формула $P(t_1 \dots t_n)$, P – предикативный символ арности n; t_i – термы
- $$\begin{aligned} \neg A \\ A \vee B \\ A \wedge B \\ A \supset B \end{aligned}$$

  правильные формулы
- Если F – правильная формула со свободной x то, $(\forall x)F$ и $(\exists x)F$ – правильные со связанной x

I. Вопрос 3.3

3. **Множество аксиом** Из логики высказываний:

$$\begin{aligned} p &\supset (q \supset p) \\ (p \supset (q \supset r)) &\supset ((p \supset q) \supset (p \supset r)) \\ (\neg p \supset \neg q) &\supset (q \supset p) \end{aligned}$$

Новые:

$$\begin{aligned} (\forall x)A(x) &\supset A(t) \\ A(t) &\supset (\exists x)A(x) \end{aligned}$$

4. **Правила вывода**

- Из логики высказываний (*modus ponens.*):

$$A, A \supset B \vdash B$$

- Новые (A не содержит свободных вхождений переменную x):

$$A \supset B(x) \vdash A \supset (\forall x)B(x)$$

$$B(x) \supset A \vdash (\exists x)B(x) \supset A$$

- Правила замены (предикативная переменная на предикат)

I. Вопрос 4.1

4 Понятие о полноте

Понятие о полноте, непротиворечивости и корректности логической системы. Связь логического вывода с общезначимостью. Теорема о дедукции.

Полнота

Определение 13. *Формула F называются* ***общезначимой*** $\models F$ *если она истина во всех интерпретациях.*

Определение 14. *Формула F называются* ***выполнимой*** *если она истина хотя бы в одной интерпретации.*

Определение 15. ***Полное исчисление*** *– то в котором любая общезначимая формула выводима*

$$\models A \Rightarrow \vdash A$$

Непротиворечивость

1. ***Формальная*:**
$$\nexists A((\vdash A) \wedge (\vdash \neg A))$$
2. ***Семантическая* (*достоверность*):**
$$\vdash A \Rightarrow \models A$$

Теоремма 4 (О Дедукции)**.** •

- *Для высказываний:*
$$\Gamma, A \vdash B \Leftrightarrow \Gamma \vdash A \supset B$$
где Γ произвольное множество логики высказываний На основе теоремы:
$$\models A \Leftrightarrow \vdash A$$
- *Для предикатов:*
Если $\Gamma, A \vdash B$, при этом в процессе вывода не участвуют свободные переменные из A, то $\Gamma \vdash A \supset B$

  Теоремма 5 (Гёделя о полноте)**.**
$$\models A \Rightarrow \vdash A$$
  Любая общезначимая формула выводима.

I. Вопрос 5.1

5 Алгоритмическая неразрешимость исчисления предикатов

Алгоритмическая неразрешимость исчисления предикатов. Теорема Черча. Теорема Геделя о неполноте формальной арифметики.

Полнота

Теоремма 6 (Гёделя о полноте)**.**

$$\models A \Rightarrow \vdash A$$

Любая общезначимая формула выводима.

Теоремма 7 (Гёделя о не полноте)**.** *Любая прикладная теория первого порядка, содержащая формальную арифметику, не является полной теорией.* $\exists$ *такие истинные формулы* $\models F$*, что ни* F *ни* $\neg F$ *не являются выводимыми, в соответствующем исчислении.*

Невозможно полностью исследовать реальность формальными методами.

Определение 16. *Разрешимое исчисление – в котором,* $\exists$ *алгоритм проверки выводимости произвольной формулы в этом исчислении, т.е. алгоритм, который по произвольной формуле за конечное время выдаст соответствующий результат.*

Методы:

- Синтаксический (построение логического вывода)
- Семантический (таблицы истинности)

Теоремма 8 (Чёрча)**.** *Исчисление предикатов является неразрешимой теорией, т.е. не существует алгоритма установки общезначимости произвольной формулы логики предикатов.*

Если некоторая формула выводима в исчислении предикатов то факт ее выводимости можно установить при помощи перебойного алгоритма(Строим цепочки всевозможных выводов). За конечное время установим цепочку вывода для общезначимой формулы. Для **не общезначимой** опровергнуть ее общезначимость не удастся.

I. Вопрос 6.1

6 Нормальные формы

Нормальные формы (КНФ, Скулемовская и клаузальная). Дизъюнкты Хорна. Сведение формул логики предикатов к фразам Хорна.

Определение 17. *Литера – атомарная формула или ее отрицание*

Определение 18. *Дизъюнкт – формула из дизъюнкции литер*

Форма	Вид	Приведение
КНФ	Конъюнкция элементарных дизъюнкции	Дистрибутивность $A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$ де Морган $\neg(A \vee B) \equiv \neg A \wedge \neg B$ $\neg(A \wedge B) \equiv \neg A \vee \neg B$ Определения $(A \supset B) = \neg A \vee \neg B$ $(A \equiv B) = (A \wedge B) \vee (\neg A \wedge \neg B)$
Предваренная (ПНФ)	$(\heartsuit x_1 \dots \heartsuit x_n) F(x_1 \dots x_n)$ $\heartsuit \in \{\exists, \forall\}$ F не содержит кванторов	1. Перенос кванторов в начало формулы 2. Устранение конфликтов между именами связанных переменных
Скулемова (СНФ)	Квантор $\exists$ удаляется, а соответствующая переменная заменяется на константу.	• Выберем самый левый $\exists$ • заменяем $\exists$ со связанной переменной на не входящий в формулу функциональный символ • Пока остался хотя бы 1 $\exists$ повторяем алгоритм

I. Вопрос 6.2

Клаузальная (КлНФ)	Дизъюнкция литер $A_1 \vee A_2 \cdots \vee A_n;$ $A_i \in \{P(), \neg P()\};$	1. Приведение к ПНФ 2. Приведение к СНФ 3. Отбрасывание $\forall$ (для компактности) 4. Приведение к КНФ 5. Запись дизъюнктов в виде множества формул в КлНФ
Дизъюнкт Хорна	$A \vee \neg B_1 \cdots \vee \neg B_n;$	$A \vee \neg B_1 \vee \cdots \vee \neg B_n$ $=$ $\neg B_1 \vee \cdots \vee \neg B_n \vee A$ $=$ $\neg(B_1 \wedge \cdots \wedge B_n) \vee A$ $(\neg X \vee Y = (X \supset Y))$ $=$ $(B_\wedge \cdots \wedge B_n) \subset A$ Если пройти в обратном порядке, получаем сведение формул логики предикатов к дизъюнкту Хорна. Основное ограничение: нет отрицания($\neg$) в посылках и заключениях.

I. Вопрос 7.1

7 Унификация

Унификация. Правила унификации сложных структур.

Определение 19. *Подстановка*

$$\Theta = \{X_1/t_1, \dots X_n/t_n\}$$

где

$$(\forall i \neq j) X_i \neq X_j, t_i \neq X_i$$

.

Определение 20. *Применение* Θ *к формуле или терму* F *(*$F\Theta$ *) – Формула или терм (опр. рекурсивно), где все* X_i *заменены на* t_i*.*

Определение 21. *Унификатор – подстановка* Θ *, для формул* F *и* G*, если* $F\Theta = G\Theta$*.*

Определение 22. *Угифицирующиеся – формулы, для которых* $\exists$ *унификатор.*

Смысл унификации – выявлять эквивалентные формулы, которые могут быть приведены одна к другой путем замены переменных.

Если множество унифицируемо, то существует, как правило, не один унификатор этого множества, а несколько. Среди всех унификаторов данного множества выделяют так называемый наиболее общий унификатор.

Не из лекций:

> Пусть $\eta = \{x_1/t_1, \dots, x_k/t_k\}$ и $\xi = \{y1/s1, \dots, y_l/s_l\}$ – две подстановки. Тогда **произведением подстановок** η и ξ называется подстановка, которая получается из последовательности равенств.
>
> $$\{x_1/\xi(t_1), \dots, x_k/\xi(t_k), y_1/s_1, \dots, y_n/s_n\}$$
>
> вычеркиванием равенств вида $x_i = x_i$ для $1 <= i <= k$, $y_j = s_j$, если $y_j \in x1, \dots, xk$, для $1 <= j <= l$.
>
> Для обозначения результата действия подстановки на дизъюнкт мы применяем префиксную функциональную запись, поэтому произведение подстановок η и ξ будем обозначать через и $\eta \circ \xi$, подчёркивая тем самым, что сначала действует η, а потом ξ.
>
> Для любых подстановок η, ξ, ϑ
>
> $$\eta \circ (\xi \circ \vartheta) = (\eta \circ \xi) \circ \vartheta$$

I. Вопрос 7.2

Определение 23. *Унификатор σ множества литералов или термов называется* ***наиболее общим унификатором*** *этого множества, если для любого унификатора τ того же множества литералов существует подстановка ξ такая, что $\tau = \xi \circ \sigma$*

Или проще:

Определение 24. *Наиболее Общий Унификатор (mgu) — содержит только замены, необходимые для приведения двух формул к идентичному виду. Обозначают:*

$$\Theta = mgu(F, G).$$

7.1 Правила

1. **Переменные.** Чем угодно. При унификации другой переменной, они `связываются`
2. **Константы.** Только такой же константой.
3. **Предикаты.** Должны совпасть:
   (a) функциональный символ
   (b) арность
   (c) все аргументы попарно унифицируются

   Иначе унификации нет.

I. Вопрос 8.1

8 Правило вывода *modus tollens*

Правило вывода *modus tollens*. Простое и обобщённое правило резолюции.

8.1 *modus tollens*

$$A \subset B, \neg B \vdash \neg A$$

Если доказательство $A \subset B$ верно,
но $\neg B$,
то $\neg A$

8.2 *Резолюция*

- Простая

$$\neg A \vee B, \neg B \vdash \neg A$$

- Расширенная

$$(L_1 \vee \cdots \vee L_n) \vee \neg A, (M_1 \vee \cdots \vee M_k) \vee A \vdash (L_1 \vee \cdots \vee L_n) \vee (M_1 \vee \cdots \vee M_k)$$

$$L_1 \vee \cdots \vee L_n, M_1 \vee \cdots \vee M_k$$

– литеры, A – атомарная формула.

Посылки: $(L_1 \vee \cdots \vee L_n) \vee \neg A, (M_1 \vee \cdots \vee M_k) \vee A$

Резольвента: $(L_1 \vee \cdots \vee L_n) \vee (M_1 \vee \cdots \vee M_k)$

- Обобщённая

$$\begin{aligned} (L_1 \vee \cdots \vee L_n) \vee \neg A \quad &, \quad (M_1 \vee \cdots \vee M_k) \vee A \\ \vdash ((L_1 \vee \cdots \vee L_n) \quad &\vee \quad (M_1 \vee \cdots \vee M_k))\Theta; \\ \Theta = mgu(A, B); & \end{aligned}$$

Очень мощное правило. Достаточно для построения дедуктивной системы. Достигается за счет того, что не требуется правил подстановки, – понятие унификации позволяет автоматически находить подходящие правила, применения резолюций.

Для доказательства методом резолюций используется метод опровержения.

`Надо доказать A из посылок` $\{B_1, B_n\}$
`Пытаемся доказать противоречивость` $\{B_1, B_n, \neg A\}$

I. Вопрос 8.2

При помощи метода резолюций пытаются получить пустой дизъюнкт $\square$

$$A, \neg A \subset \square$$

I. Вопрос 9.1

9 Метод резолюции

Метод резолюции для реализации эффективного логического вывода в логических программах на основе фраз Хорна. Стратегии резолюции. SLD-резолюция.

9.1 Стратегии резолюции

Определение 25. *Стратегии резолюции – метод, в соответствии с которым происходит выбор двух формул, подлежащих резолюции.*

Стратегии:

- **Определение 26.** *Линейная (L-резолюция) – на каждом шаге используется формула, полученная на предыдущем.*

  Для первого шага в качестве посылки берётся добавленное к множеству формул отрицание доказываемого предположения.

- **Определение 27.** *С выбирающим правилом (S-резолюция) – литера для применения выбирается в соответствии с некоторым правилом.*

- **Определение 28.** *Упорядоченная S-резолюция (R-резолюция) – важен порядок литер в дизъюнктах*

  В качестве общего дизъюнкта используют самую левую литеру.

При SL и R -резолюциях на каждом шаге одна из посылок применения нам известна. При реализации процедуры логического вывода на компьютере удаётся избежать лишнего перебора всех формул, а так же позволяет избавиться от перебора всех литер данного дизъюнкта. Двойной перебор → один.

SL-резолюция:

- +:
  Эффективность
- −:
  Ограничение общности

⇒ Вопрос полноты ⇒ Полнота достигается на дизъюнктах Хорна. Соответствующая резолюция – SLD-резолюция.

I. Вопрос 9.2

9.2 SLD-резолюция

Определение 29. *SLD-резолюция – разновидность метода линейной резолюции, в котором для выбора второй посылки при выполнении резолютивного вывода используется некоторая детерминированная функция.*

Определение 30. *Дизъюнкт Хорна*

$$A \vee \neg B_1 \cdots \vee \neg B_n;$$

$$\begin{aligned}
A &\vee \neg B_1 \vee \cdots \vee \neg B_n \\
&= \\
\neg B_1 \vee \cdots \vee \neg B_n &\vee A \\
&= \\
\neg(B_1 \wedge \cdots \wedge B_n) &\vee A \\
(\neg X \vee Y &= (X \supset Y)) \\
&= \\
(B_\wedge \cdots \wedge B_n) &\subset A
\end{aligned}$$

Если пройти в обратном порядке, получаем сведение формул логики предикатов к дизъюнкту Хорна. Основное ограничение: нет отрицания(¬) в посылках и заключениях.

I. Вопрос 10.1

10 Рекурсивное описание алгоритма работы SLD-резолютивного логического интерпретатора

Рекурсивное описание алгоритма работы SLD-резолютивного логического интерпретатора. Механизм бэктрекинга. Дерево вывода. Стратегии обхода дерева вывода при поиске решений.

Определение 31. *SLD-резолюция – разновидность метода линейной резолюции, в котором для выбора второй посылки при выполнении резолютивного вывода используется некоторая детерминированная функция.*

Рекурсивное описание:

1. Добавляем к множеству фактов отрицание цели.
2. Ищем формулу, такую, позитивная литера которой унифицируется с целью.
3. Применяем резолюцию (*modus tollens*) для каждой удачной унификации:
   - если удалось получить пустой дизъюнкт $\Rightarrow$ завершить.
   - иначе продолжить.
4. Применяем этот же алгоритм. Цель = формула найденная на предыдущем шаге.

Процесс можно представить графически в виде дерева вывода. (стр. 61 в книжке). Это подчёркивает рекурсивную структуру SLD-вывода. При поиске решения происходит исследование ветвей дерева.

Определение 32. *Дерево вывода – дерево, в котором развилки возникают только в случае множественных вариантов доказательства, шаги конъюнкции – линейная последовательность вершин.*

В общем случае SLD-резолюция не должна быть упорядочена, но стратегия выбора должна задаваться **выбирающим правилом**.

10.1 Стратегии обхода дерева вывода при поиске решений.

- Обход в глубину
  от корня по некоторому направлени я, пока поиск не дойдёт до листа. В случае необходимости поиска другого решения происходит откат назад (вверх) на 1 шаг – **backtracking**, и исследование других возможных вариантов решения.
- Обход в ширину
  на каждом уровне одновременно рассматриваются все возможные варианты. Наиболее короткие пути найдутся быстрее.

I. Вопрос 11.1

11 Декларативная и процедурная семантика языка логического программирования

Декларативная и процедурная семантика языка логического программирования. Примеры.

Семантика логических программ может отличаться от исходной семантики логических формул. Логически эквивалентные программы могут выполняться по-разному.

Семантика:

- **Декларативная**
  - Программа – множество формул логики предикатов.
  - Решение – логические следствия формул.

  Учёт: позволяет решить задачу.
  Правильная программа: может работать не эффективно.
- **Процедурная** задаётся алгоритмом SLD-вывода и алгоритмом поиска. Учитывается процесс вывода.
  - Программа – последовательность команд для алгоритма вывода.
  - Решение – результат его работы.

  Учёт: позволяет повысить эффективность.
  Правильная программа: может быть бессмысленной.
  Рекомендации:
  1. Условие окончании рекурсии на 1 месте
  2. В рекурсивном определении, сначала не рекурсивные предикаты.

I. Вопрос 11.2

Решение проблемы бесконечной рекурсии в различных языках:

- **ProLog** Никак. Приходится использовать процедурные ухищрения. *Пример:* Предикат предок:

$$(\forall X)(\forall Y)(((\exists Z)A(X,Z) \wedge P(Z,Y)) \vee P(X,Y)) \subset A(X,Y)$$

  Приедем к Хорну:

  - **Плохо**

$$\begin{array}{lcl} A(X,Y) & \vee & \neg A(X,Z) \vee \neg P(X,Z) \\ A(X,Y) & \vee & \neg P(X,Y) \end{array}$$

    Дерево с бесконечной левой веткой.

  - **Хорошо**

$$\begin{array}{lcl} A(X,Y) & \vee & \neg P(X,Y) \\ A(X,Y) & \vee & \neg A(X,Z) \vee \neg P(X,Z) \end{array}$$

    Условие окончании рекурсии на 1 месте.

  Факты:

$$\begin{array}{l} P(a,b) \\ P(a,c) \\ P(c,d) \end{array}$$

  `НЕ` Цель:

$$\neg A(U,d)$$

- **Mercury** Автоматическая оптимизация процесса вывода. Сложное выбирающее правило. Выбираются предикаты приводящие к более ранней конкретизации переменных. Программист должен описать максимально возможные режимы вызова предиката.

- **DataLog** Поиск в ширину.

12 Отрицание в логическом программировании

Отрицание в логическом программировании. Отрицание по неуспеху и предположение о замкнутости мира. SLDNF-резолюция.

В логических программах невозможно эффективно реализовать отрицание. ⇒ Рассматривают нечто подобное по семантике.

Определение 33. ***Отрицание по неуспеху*** *— отрицание некоторого утверждения P,*

- *истина, $\Leftarrow P$* ***не*** *выводится при помощи доступных механизмов вывода.*
- *ложь, $\Leftarrow P$ выводится при помощи доступных механизмов вывода.*

Каждое выражение стоящее под знаком отрицания требует отдельного доказательства, которое не связано с текущим контекстом унификации (Лес). Отрицание по неуспеху ≡ логическому отрицанию, только в предположении о замкнутости мира (CWA):

$$(\forall A) \vdash (\vdash \neg A)$$

отрицание по бесконечному неуспеху

12.1 SLDNF-резолюция

```
1  p :- p.
2
3  q :- q.
4  q.
```

Попытки доказать эти предикаты приводят к бесконечному дереву вывода. Из CWA $\neg p$ оказывается не выводим. ⇒ Введём более бедное **отрицание по конечному неуспеху**

$$(\vdash \neg A) \Leftrightarrow (\exists N) n <= N$$

, где n глубина SLD-дерева неуспешного вывода.

Определение 34. ***Дополненная логическая программа*** *$comp(\rho)$ — для программы $\rho \subset$ в предикатах заменено на $\equiv$ в соответствии с CWA.*

Теоремма 9 (О достоверности и полноте $\neg$ по конечному неуспеху)**.** *Пусть ρ логическая программа без отрицания. Некоторое утверждение A имеет конечное SLD-дерево вывода, заканчивающегося неуспехом $\Leftrightarrow comp(\rho) \models \neg A$.*

Определение 35. ***Обобщённая логическая программа*** *— содержит $\neg$ в посылках правил.*

I. Вопрос 12.2

Алгоритм вывода ОЛП (**SLDNF-резолюция**) дополняет SLD-резолюцию:

- $\neg A$ закончится успехом $\Leftrightarrow \exists$ конечное дерево **неуспешного** вывода для A.
- $\neg A$ закончится **не** успехом $\Leftrightarrow \exists$ конечное дерево **успешного** вывода для A

Процесс вывода представляет из себя лес. По одному дереву на каждое выражение под знаком отрицания.

- Свойство полноты не выполнено
- Свойство достоверности выполняется не всегда (не выполненное $A \equiv \neg\neg A$),
- Нарушается чистота логической программы (различные ответы при перестановке предикатов)
- Нельзя использовать предикаты с отрицание для генерации значений

$$\neg(\exists X)(\exists Y)parent(X, Y) \equiv (\forall X)(\forall Y)\neg parent(X, Y)$$

12.2 Предикат *not*

```
1  not(Predicate):-
2    Predicate,!,fail.
3  not(_).
```

II Язык программирование Prolog

13 Пролог как язык логического программирования

Пролог как язык логического программирования.

Логическая программа:

- Декларативная часть – набор предложений, описывающий предметную область. Подобный подход берет начало в математической логике.
  - факты
  - правила
- Запросы – ответы на которые – результат. Выполнение – поиск значений переменных, которые удовлетворяют условиям. Множество `всех` решений ищется системой на основе backtracking'а

$\exists$ множество систем логического программирования, большинство из них использует Пролог.

13.1 Основные смысловые объекты программы

Термы:

- Простые
  - Переменные
  - Константы
    - Атомы
    - Числа
- Структурные

Лексику Пролога можно описать грамматикой:

```
    терм      ::= константа
           | функтор"(" терм {","терм}")"
           | оператор терм
           | терм оператор
           | терм оператор терм
           | "(" терм ")".
константа ::= атом | число.
оператор  ::= функтор.
функтор   ::= атом.
```

II. Вопрос 13.2

Унификация:

- Свободная переменная – с произвольным термом и связывается
- Связанная переменная – как значение с которым она связана
- Функциональный терм Должны совпасть:
    1. функциональный символ
    2. арность
    3. все аргументы попарно унифицируются
- Константа – с такой же константой.

II. Вопрос 14.1

14 Основные объекты языка Пролог

Основные объекты языка Пролог: атомы, числа, константы, переменные, термы, структурные термы, предикаты.

Термы:

1. **Простые**

   (a) **Переменные** (`[A-ZA-Я].*`) *Область действия* – 1 правило.
   При интерпретации заменяются внутренними ссылками.

      i. **Анонимные** (`[_]`) Каждое вхождение обозначает новую переменную.

```
1  has_child(X):− parents(X, Y, Z).
2  has_child(X):− parents(Y, X, Z).
```

⇕

```
1  has_child(X):− parents(X, _, _).
2  has_child(X):− parents(_, X, _).
```

   - Связанные
     - Переменная связывается в результате унификации.
     - Значение меняется только при backtracking.
   - Свободные Подразумевается изначально.

   (b) **Константы**

      i. **Атомы** Константы предметной области. 1 идентификатор ↔ 1 объект.
         - `[a-za-я].*`.
         - `[^\<=>\-/]+` спецсимволы.
         - `['].*?[']` ∀ в апострофах.

      ii. **Числа** (`[0-9]+`) Введены в явном виде исключительно для удобства. Можно обойтись и без них, построив формальную арифметику на аксиомах Пеано.

II. Вопрос 14.2

2. **Структурные** $f(t_1, \ldots t_n)$, f – функтор арности n. Обозначение: f/n.
Используют: сложные факты

```
1    появление('Тоня Глоткина', дата(11, июль, 1989))
```

Графика: в виде дерева

- появление
  - 'Тоня Глоткина'
  - дата
    - 11
    - июль
    - 1989

Возможны рекурсивные определения. Предикаты – тоже структурные термы.

II. Вопрос 15.1

15 Операторы

Операторы. Представление и вычисление арифметических выражений. Определяемые пользователем операторы. Представление предикатов и правил структурными термами.

15.1 Операторы

Для чего нужны: естественная, удобная запись арифметических выражений, использование инфиксной и постфиксной записи вместо префиксной.

Свойства:

- Учёт приоритета
- Группировка скобками
- Ассоциативность
  - Левая $(+, -, *, /)$
  - Правая (степень)

15.2 Представление и вычисление арифметических выражений

- $=$, создаёт структурный терм Интересный пример символьными вычислениями (стр. 88).
- is, вычисляет, правая часть должна быть правильным арифметическим выражением, из констант может содержать только числа

15.3 Определяемые пользователем операторы

Для объявления оператора используется встроенный системный предикат *op*.
Формат вызова:

```
1   :- op(<приоритет>, <шаблон>, <оператор>)
```

- **Приоритет** $\in [0, 255]$ (в классике). Чем больше число тем **меньше** реальный приоритет. У `:-` он 255.
- **Шаблон** задаёт:
  - арность
    - унарный
      `fx; fy; xf; yf;`
    - бинарный
      `yfx; xfy; xfx;`
  - позиционность
    - префиксный
      `xf; yf;`
    - инфиксный
      `yfx; xfy; xfx;`
    - постфиксный
      `fx; fy;`
  - ассоциативность если нет скобок, допускает:
    x – оператор более высокого приоритета
    y – оператор более высокого или равного приоритета
    - левая

      `yfx;`
    - правая
      `xfy;`
    - никакая
      `xfx; xf; fx;`
    - просто она есть, какая – не определено
      `fy; yf;`
- **Оператор**
  - последовательность спецсимволов (`<==>`).
  - функциональный символ (`seq`).

15.4 Представление предикатов и правил структурными термами

Элементы пролог программы такие как, предикаты, и описывающие их

- Факты
- Правила

являются структурными термами

```
1  resistance(par(X, Y), R) :−
2    resistance(X, RX),
3    resistance(Y, RY),
4    R is RX ∗ RY / (RX + RY).
```

В префиксной нотации (lisp forewer):

```
1  :− ( resistance(par(X, Y), R),
2       , (
3         , (
4             resistance(X, RX),
5             resistance(Y, RY)
6           ),
7           is (
8              R,
9             /(
10              ∗(RX , RY)
11              +(RX , RY)
12            )
13            )
14        )
15    ).
```

16 Подходы к организации циклов в логическом программировании

Подходы к организации циклов в логическом программировании. Примеры. Предикаты *for*, *repeat*.

Подходы:

- **рекурсия** На каждом шаге создаётся новая среда, новое значение инварианта цикла.

```
1   print(11).
2   print(X) :−
3     X =< 10, write(X), X1 is X + 1, print(X1).
4   print :−
5     print(1).
```

Бесконечный цикл:

```
1   iloop :−
2     iloop
```

Современные системы логического программирования умеют распознавать хвостовую рекурсию.

- **возврат** – циклы управляемые неуспехом. Новое значение переменной цикла можно генерировать каждый раз в процессе возврата.

```
1   print :−
2     gen(1, X), write(X), fail.
3   gen(X, X).
4   gen(Y, X):−
5     Y < 10, Y1 is Y + 1, gen(Y1, X).
```

Бесконечный цикл:

```
1   iloop :−
2     loop, fail
3   loop.
4   loop :− loop.
```

II. Вопрос 16.2

16.1 *repeat.*

Возвратный цикл с постусловием. `repeat` всегда успешно доказуем. В процессе каждого возврата он генерирует новое решение, не позволяя перешагнуть через себя.

```
1    repeat.
2    repeat :- repeat.
```

Использование:

```
1    read_dim:-
2      repeat, read(X), X >= 3, X <= 6.
3
4    read_ans:-
5      repeat, read(X), member(X, [yes, no]).
```

Вообще говоря, это работает только с пердикатами, у котороых есть побочный эффект. С классическими предикатами, `repeat` будет возвращаться один и тот же ответ.

16.2 *for.*

```
1    for(A, A, _).
2    for(X, A, B):-
3      A < B, A1 is A + 1, for(X, A1, B).
```

Вариант другой, хуже, но можно его идею с (`member`) для организации цикла как в Python, (цикл по произвольному множеству).

```
1    integer(A, A, [A]).
2    integer(A, B, [A|R]):-
3      A < B, A1 is A + 1,
4      integer(A1, B, R).
5
6    for(X, A, B):-
7      integer(A, B, L), member(X, L).
```

Использование:

```
1    ?- for(X, 1, 10), write(X), fail.
```

17 Управление перебором и отсечение

Управление перебором и отсечение(*cut*). Предикат *not*.

Отсечение обладает только процедурной семантикой. Отсечение приводит к тому, что сделанный выбор фиксируется и при откате назад альтернативные решения не учитываются.

Определение 36. ***Родительское целевое утверждение*** *— целевое утверждение, которое приводит к вызову правила, содержащего отсечение.*

Процедурная семантика отсечения:

> Встречая отсечение, процессор логического вывода
> не меняет решения принятое с момента вызова
> родительского целевого утверждения.

```
1  C:− P, Q, !, R, S.
2    % R1 % РОДИТЕЛЬСКОЕ ЦЕЛЕВОЕ УТВЕРЖДЕНИЕ
3  C:− T.
4    % R2
5  A:− B, C, D.
```

При попытке доказательства выполнится R1:

- $\exists(P, Q) \Rightarrow$ выполнится *!*, фиксируются:
  - Решения (P, Q).
  - Выбор R1.

  $\Rightarrow$ R2 не выполнится и не будет произведен backtracking для поиска других P, Q. Правила R, S разбираются на общих основаниях.
- $\nexists(P, Q)$.

II. Вопрос 17.2

Отсечения(тип зависит от контекста вызова предиката):

- Зелёное. Нет потери решений.
- Красное. Потеря решений.

Варианты использования отсечения:

- Минимизация перебора (для оптимизации).

```
1    father(X, Y):− parent(X, Y), male(X).
```

⇳ У человека 1 отец. Не надо проверять остальные варианты.

```
1    father(X, Y):− parent(X, Y), male(X), !
```

- Для выбора отдельных решений.

```
1    numOfLegs(cetipede, X):− !, X = 40.
2    numOfLegs(human, X):− !, X = 2.
3    numOfLegs(_, 4).
```

- Отрицание по неуспеху. `!-fail` Отрицание может быть определено через отсечение.

```
1    not(Predicate):−
2      Predicate,!,fail.
3    not(_).
```

- Условный оператор.

```
1    Q :− if A then B, else C.
```

⇳

```
1    Q :− A, !, B.
2    Q :− C.
```

- Удаление нежелательных вариантов.

```
1    fact(1,1):− !.
2    fact(N,F):− N1 is N − 1,
3      fact(N1,F1), F is F1∗N.
```

18 Встроенные предикаты Пролога: read, write, nl

Встроенные предикаты Пролога: read, write, nl.

Для классических языков логического программирования, с запросным режимом работы, операции ввода-вывода не являются необходимыми, но иногда они оказываются очень полезными.

Основные предикаты ввода-вывода в Прологе:

- `write(X)`
  Выводит занчение X на stdout.
- `read(X)`
  Вводит терм(с точкой на конце) с stdin и унифицирует его с X.
- `nl`
  Эквивалентен `write("\n")` .

Примеры:

```
1  main :-
2    specialty(tonja, X),
3    write(X), nl, fail.
```

```
1  main :-
2    wrire('Имя_студентки'),
3    read(Name), % работет единожды в отличие от write
4    specialty(Name, Spec),
5    write(Spec), nl, fail.
```

Работа с файлами (стр 94):

- `see(fileName)`
- `tell(fileName)`
- `seeing(fileName); seeing(X)`
- `telling(fileName); telling(X)`

19 asserta/assertz, retract, findall, bagof/setof

Предикаты динамического изменения базы данных Пролога:
`asserta/assertz, retract`. Примеры. Предикаты поиска множества решений:
`findall, bagof/setof`.

С http://www.webauto.ru/forum08/ (Древний, мертвый форум)

19.1 asserta/assertz, retract

В базу знаний(фактов и правил) Пролога можно добавлять факты и правила прямо по ходу выполнения программы. Это осуществляется при помощи предикатов

- `assert` Как предусмотрено в большинстве реализаций Пролога, эквивалентен `assertz`.
- `asserta` Добавляет факт/правило в начало базы.
- `assertz` Добавляет факт/правило в конец базы.
- `retract` Удаление факта.

Для того, чтобы добавить факт или правило в базу знаний, необходимо написать, например, следующее:

```
1  assert( fact( a ) )
2  % или
3  assert( rule( X ):- fact( X ) )
```

Из следующего примера видно, как данные предикаты оказывают влияние на результаты запросов:

```
1  ?- assert( fact( b ) ), assertz( fact( c ) ), assert(fact( d ) ), asserta( fact( a ) ).
2    Yes.
3  ?- fact( X ).
4    X = a;
5    X = b;
6    X = c;
7    X = d;
8    No.
```

Описываемые предикаты могут применяться в случаях, когда необходимо сохранить ответы, на заданные пользователю вопросы в ходе работы программы (как например в экспертных системах), или, например, для создания аналога глобальных переменных.

19.2 findall, bagof/setoft

При помощи добавления и удаления фактов в и из базы знаний реализованы предикаты поиска множества решений findall/bagof и setof. Предикаты findall и bagof имеют три аргумента:

1. `findall(X,pred(X,...),L).`
   - `X` — переменная, значения которой нас интересуют
   - `pred(X,...)` — предикат, в результате работы которого X преобретает значения, у этого предиката может быть больше одного аргумента.
   - `L` — список всех значений X, котороые можно получиь в результате работы pred.

```
1   findall(X,P,L):-
2       (
3           call(P), % Вызов предиката P, поиск решения
4           assertz(found(X)), % Запись решения в базу знаний
5           fail % Инициация неуспеха для backtracking
6       );
7       collect(L).
8
9   collect(L):-
10      (
11          retract(found(X)), % Получение текущего решения
12          L=[X|Rest], % Добавление его в список
13          collect(Rest) % Сбор остальных решений
14      );
15      L=[]. % Решения кончились
```

2. `bagof(X,pred(X,...),L).`
   работает и выглядит аналогично `findall`, но нет автоматической связки переменных квантором существования.

3. `setof(X,pred(X,...),L).`
   отличается от них тем, что возвращает отсортированный список решений, в котором удалены все повторения.

II. Вопрос 19.3

Пример с http://www.csupomona.edu/.../prolog_tutorial/

```
1  p(1,3,5).
2  p(2,4,1).
3  p(3,5,2).
4  p(4,3,1).
5  p(5,2,4).
6
7  ?- bagof(Z,p(X,Y,Z),Bag).
8  Z = _G182 X = 1 Y = 3 Bag = [5] ;
9  Z = _G182 X = 2 Y = 4 Bag = [1] ;
10 Z = _G182 X = 3 Y = 5 Bag = [2] ;
11 Z = _G182 X = 4 Y = 3 Bag = [1] ;
12 Z = _G182 X = 5 Y = 2 Bag = [4] ;
13 No
14
15 ?- findall(Z,p(X,Y,Z),Bag).
16 Z = _G182 X = _G180 Y = _G181 Bag = [5, 1, 2, 1, 4] ;
17 No
18
19 ?- bagof(Z,X^Y^p(X,Y,Z),Bag).
20 Z = _G182 X = _G180 Y = _G181 Bag = [5, 1, 2, 1, 4] ;
21 No
22
23 ?- setof(Z,X^Y^p(X,Y,Z),Bag).
24 %$$
25 % (\exists X)(\exists Y)p(X, Y, Z)
26 %$$
27 Z = _G182 X = _G180 Y = _G181 Bag = [1, 2, 4, 5] ;
28 No
29
30 ?- bagof(Z,(p(X,Y,Z),Z>5),Bag).
31 No
32
33 ?- findall(Z,(p(X,Y,Z),Z>5),Bag).
34 Z = _G182 X = _G180 Y = _G181 Bag = []
35 Yes
```

III Рекурсивные структуры данных

20 Представление списков

Представление списков. Связь списков и структурных термов. Оператор =..

Аппарат логики предикатов содержит в себе все необходимое для представления списков (потенциально бесконечных). Для этого можно использовать структурные термы, по аналогии с аксиомами Пеано для формальной арифметики.

Определение 37. ***Список*** —

- *пустой список* $\varnothing$
- *структурный терм* $\Lambda(t, \zeta)$
  - Λ – *имя*
  - t – *произвольный терм (голова)*
  - ζ – *список (хвост для* $\Lambda(t, \zeta)$*)*

`Список` $\equiv$ `Дерево`.

Построение:

- `[]` – Пустой список.
- .:
$$\Lambda(1, \Lambda(2, \Lambda(3, \varnothing))) \equiv .(1, .(2, .(3, [])))$$
- Скобочная нотация:
$$\Lambda(1, \Lambda(2, \Lambda(3, \varnothing))) \equiv [1, 2, 3]$$

Унификация, для отделения головы и хвоста:

- `[]`: не определена.
- `[ОдинокийЭлемент]`: Хвост = [].
- .:

```
1    .(Голова, Хвост) = список.
```

- Скобочная нотация:

```
1    [Голова | Хвост] = список.
```

– Несколько головных:

$$[A, B|T] \equiv .(A, .(B, T))$$

– Построение, из головы и хвоста

Возможны многоуровневые списки `[[1, [2]], 3, [4]]` – сильнее подчёркивает связь с деревом.

Для декомпозиции термов и создания новых термов предусмотрены три встроенных предиката: `functor`, `arg` и `=..`. Предикат `=..`, который записывается как инфиксный оператор и читается как «юнив» (univ).

```
1   Term =.. L
```

является истинной, если L — список, содержащий главный функтор `Term`, за которым следуют его параметры. Применение этого предиката демонстрируется на следующих примерах:

```
1  ?- f ( a, b) =.. L.
2  L = [f, a, b]
3  ?- T =.. [ rectangle , 3, 5].
4  T = rectangle( 3, 5)
5  ?- Z =.. [p, X, f (X, Y)] .
6  Z = p( X, f(X,Y) )
```

Преимущество этого подхода состоит в том, что программа приобретает способность в процессе своей работы самостоятельно вырабатывать и выполнять цели, представленные в формах, которые не всегда можно было предвидеть ко времени написания этой программы.

III. Вопрос 20.3

Применение:

- Обобщённые вычисления, в обход требований синтаксиса.

```
1     enlarge( Fig, F, Figl) :−
2       Fig =.. [Type | Parameters],
3       multiplylist( Parameters, F, Parameters1 ) ,
4       Figl =.. [Type | Parametersl ].
5
6     multiplylist( [], _, [] ).
7       multiplylist( [X | L], F, [X1 | L1]) :−
8       X1 is F ∗ X, multiplylist( L, F, L1 ).
```

- Символьные вычисления, в обход требований синтаксиса.

```
1           % substitute ( Subterm, Term, Subterml, Terml) :
2           %   если все вхождения субтерма Subterm
3             % в терме Term будут заменены
4           %   субтермом Subterml, то будет получен терм Terml
5
6           % Случай 1. Замена всего терма
7           substitute{ Term, Term, Term1, Term1) :− !.
8
9           % Случай 2. Ничего не требует замены,
10            % если Term относится к типу 'atomic'
11          substitute( _, Term, _, Term) :−
12            atomic(Term) , !.
13
14          % Случай 3. Выполнение замены Б параметрах
15          substitute( Sub, Term, Sub1, Term1) :−
16            Term =.. [F|Args], % Получить параметры
17            substlist( Sub, Args, Sub1 Args1), % Выполнить в них замену
18            Terml =.. [F|Args1].
19
20          substlist( _, [], _, []) .
21          substlist( Sub, [Term|Terms], Subl, [Terml|Terms1]) :−
22            Substitute( Sub, Term, Sub1, Term1),
23            Substlist( Sub, Terms, Sub1, Terms1).
24
25  ?− substitute( sin(x), 2∗sin(x)∗f(sin(x)), t, F).
26  F = 2∗t∗f(t)
```

21 Описание основных предикатов обработки списков: определение длины, взятие n-ого элемента, принадлежность элемента списку, конкатенация списков.

Описание основных предикатов обработки списков: определение длины, взятие n-ого элемента, принадлежность элемента списку, конкатенация списков.

21.1 определение длины

```
% Prolog
length([], 0).
length([_|T], N):-
  length(T, N1), N is N + 1.
```

```
:- func length(list(T)) = int.
:- mode length(in) = out is det.
length([]) = 0.
length([_|T]) = length(T) + 1.
```

21.2 взятие n-ого элемента

```
getN([Elem|_], 1, Elem).
getN([_|List], N, Elem):-
  M is N - 1,
  getN(List, M, Elem).
```

21.3 принадлежность элемента списку

```
% Prolog
member(X, [X|_]).

member(X, [_|T]):-
  member(X, T).
```

```
:- pred member(T, list(T)).
:- mode member(in, in) = is semidet.
:- mode member(out, in) = is nondet.
member(X, [X|_]).
member(X, [_|T]):- member(X, T).
```

```
member(X, L):- remove(X, L, _).
```

21.4 добавление элемента

`append(A, B, C)`, $C = A \circ [B]$

```
1  % Prolog
2
3  append([], X, [X]).
4  append([_|T], Y, [_|A]) :-
5    append(T, Y, A).
```

```
1  :- pred append(list(T), T, list(T)).
2  :- mode append(in, in, out) = is det.
3  :- mode append(out, out, in) = is multi.
4  append([], X, [X]).
5  append([_,T],Y,[_,L]):- append(T,Y,L).
```

21.5 конкатенация списков

`extend(A, B, C)`, $C = A \circ B$

```
1  % Prolog
2
3  extend([], X, X).
4  extend([_|T], Y, [_|A]) :-
5    extend(T, Y, A).
```

```
1  :- pred extend(list(T), list(T), list(T)).
2  :- mode extend(in, in, out) = is det.
3  :- mode extend(out, out, in) = is multi.
4  extend([], X, X).
5  extend([_,T],Y,[_,L]):- extend(T,Y,L).
```

21.6 применение extend

`extend(A, B, C)`, $C = A \circ B$

- Конкатенация `extend([foo1], [foo2], C)`
- Порождение разбиений `extend(X, Y, [1,2,3])`
- `rem_first(N, L, R):- extend(X, L, R), length(X, N).`
- `last(X, L):- extend(_, [X], L).`
- `next(A, B, L):- extend(_, [A,B|_], L).`
- `sublist(R, L):- extend(_, T, L), extend(R, _, T).`

22 Описание основных предикатов обработки списков: удаление элемента из списка, определение подсписка, перестановки

Описание основных предикатов обработки списков: удаление элемента из списка, определение подсписка, перестановки.

22.1 удаление элемента

```
1  remove(X, [X|T], T).
2  remove(X, [H|T], [H|R]):−
3     remove(X, T, R).
```

```
1  :− pred remove(T, list(T), list(T),).
2  :− mode remove(in,in,out) is nondet.
3  :− mode remove(out,in,out) is nondet.
4  :− mode remove(in,out,in,) is multi.
5  remove(X, [X|T], T).
6  remove(X, [H|T], [H|R])
7    :− remove(X, T, R).
```

Применение:

- Удаление (всем способами)

```
1     ?− remove(блондинка, [блондинка, брюнетка, рыжая, блондинка], X)
2     X = [брюнетка, рыжая, блондинка]
3     X = [блондинка, брюнетка, рыжая]
```

- Генерация

```
1     ?− remove(X,[бюст, талия, ноги], L)
2     X =[бюст], L =[талия, ноги];
3     X =[талия], L =[бюст, ноги];
4     X =[ноги], L =[бюст, талия];
```

`member`

```
1     member(X, L):− remove(X, L, _).
```

22.2 определение подсписка

```
1   sublist(L, R:-
2     append(_, T, L),
3     append(R, _, T).
```

22.3 перестановки

```
1  permute([], []).
2  permute(L, [X|T]):−
3    remove(X, L, R),
4    permute(R, T).
```

Мой вариант:

```
1    перемешать([X], [X]).
2    перемешать([X|L], R) :−
3      перемешать(L, R1),
4      удалить(X, R, R1).
```

```
1  :− pred permute(list(T), list(T)).
2  :− mode permute(in, in) is semidet.
3  :− mode permute(in, out) is nondet.
4  :− mode permute(out, in) is nondet.
5
6  permute([], []).
7  permute(L, [X|T]):−
8    remove(X, L, R).
9    permute(R, T).
```

Семантика

- Декларативная – симметрия.
- Императивная, если первая(вар. 1) переменная не конкретизирована – бесконечное SLD-дерево.

23 Порядковое представление списков в Прологе

Порядковое представление списков в Прологе. Представление матриц. Примеры.

Определение 38. ***Список*** —

- *пустой список* $\varnothing$
- *структурный терм* $\Lambda(t, \zeta)$
  - Λ – *имя*
  - t – *произвольный терм (голова)*
  - ζ – *список (хвост для* $\Lambda(t, \zeta)$*)*

Наряду с обычным представлением списков используется порядковое представление.

Определение 39. ***Порядковое представление*** – *такое представление списка, при котором номеру элемента соответствует его значение.*

Это очень похоже на словари в Python и C++, или на массивы в PHP. Применение:

- Матрицы. Список списков

```
1   [[101,212],[333,411]]
```

⇳

Порядковое представление

```
1   [
2     e(1,1,101), e(1,2,212),
3     e(2,1,333), e(2,2,411),
4   ]
```

23.1 Операции

1. определение длины
2. взятие n-ого элемента
3. принадлежность элемента списку
4. конкатенация списков
5. удаление элемента из списка
6. определение подсписка
7. перестановки

аналогичны как и для простых списков, но возможны более эффективные реализации. Например удаление первого элемента (`e(1, хз\_чё)`):

```
1  rem_first([], []).
2  rem_first([e(1, _)| X1 ], X2 ):−
3    rem_first(X1, X2).
4  rem_first([e(A1, B1)| X1 ], [e(A2, B2)| X2 ]):−
5    X2 is X1 − 1,
6    rem_first(X1, X2).
```

Связь порядковых и простых списков:

```
1  c2o([], []).
2  c2o(ListCommon, ListOdering):−
3          c2o_util(ListCommon, 1, ListOdering).
4
5  c2o_util([X], N, [e(N, X)]).
6  c2o_util([X|ListCommon], N, [e(N, X)|ListOdering]):−
7          N1 is N + 1,
8          c2o_util(ListCommon, N1, ListOdering).
9
10
11 ?− c2o([blonde, brunette , brown, red], X), write(X), fail.
12 [e(1,blonde),e(2,brunette),e(3,brown),e(4,red)]
13
14 ?− c2o(X, [e(1,blonde),e(2,brunette),e(3,brown),e(4,red)]), write(X), fail.
15 [blonde,brunette,brown,red]
```

24 Разностные списки

Разностные списки. Хвостовая рекурсия. Сведение рекурсивного нехвостового определения к хвостовому. Примеры.

24.1 Разностный список

Определение 40. *Разностный список* $Ł=< L_2, L_1 >$*– это пара списков* $< L_2, L_1 >$ *такая, что* L_1 *является хвостом* L_2*.*

В операторной нотации обозначают `/` или `-`.

$$L \equiv L \circ A/A$$

или

$$L \equiv L|A/A$$

24.2 Хвостовая рекурсия

Виды рекурсии (обычно реализуются с помощью стека):

- Линейная. Рекурсивный вызов – последний в определении, и нет локальных переменных требующих вычисления на дальнейших этапах.
  - Общий случай
  - Хвостовая. Реализуется без стека. Сведение к итерационному процессу.
- Нелинейная. Много(> 1) раз вызывается в теле функции. При создании двоичного дерева в императиве.
- Косвенная. Процедуры обращаются друг к другу (рекурсивный спуск)
- Хвостовая

24.3 Хвостатизация

- Ввести дополнительную переменную (аккумулятор). которая, при каждом погружении в рекурсию будет увеличиваться на 1.
- Убрать какие либо вычисления после рекурсивного вызова.

24.4 Примеры

Длинна списка

```
1  length([X], 1).
2  length([_|X], N):−
3    length(X, N1),
4    N is N1 + 1.
```

⇔

```
1  length(L, N):− length(L, 0, N).
2  length([], N, N).
3  length([X|T], S, N):−
4    S1 is S + 1,
5    length(T, S1, N).
```

Реверс

- Простой

```
1  reverse([], []).
2  reverse([X|T], R):−
3    reverse(T, T1),
4    extend(T1, [X] R).
```

- Хвостовой

```
1  reverse(L, R):− reverse(L, [], R).
2  reverse([], R, R).
3  reverse([X|T], L, R):− reverse(T, [X|L], R).
```

- Разностный

```
1  %% на самом деле не очень понятно
2  reverse(L, R):− rev(L, R/[]).
3  rev([], R/R).
4  rev([X|T], R/L):− reverse(T, R/[X|L]).
```

III. Вопрос 24.3

Конкатенация

- Простая

```
1       extend([], X, X).
2       extend([X|T], L, [X|R]):−
3         extend(T, L, R).
```

- Разностная $X - Y = (X - T) + (T - Y)$

```
1       extend(X/T, Y/T, X/Y).
2
3       ?− extend([1,2|T]/T, [3,4|Y]/Y, R).
4       R = [1,2,3,4|Y]/Y
```

25 Деревья

Деревья. Деревья поиска, основные операции с деревьями поиска.

Определение 41. ***Дерево*** —

- *пустое дерево* $\varnothing$
- *структурный терм* $\Upsilon(t, \theta)$
  - Υ – *имя терма.*
  - t – *произвольный терм (узел).*
  - θ – *множество поддеревьев.*

Деревья общего вида – родная структура в ЛП. Деревья общего вида – описывают структурные термы.
Для удобства используют двоичное дерево. В этом случае узел описывается одним и тем-же структурным термом, той же арности.

Определение 42. ***Дерево (двоичное)*** —

- *пустое дерево* $\varnothing$
- *структурный терм* $\Upsilon(t, l, r)$
  - Υ – *имя терма.*
  - t – *произвольный терм (узел).*
  - l – *левое поддерево.*
  - r – *правое поддерево.*

Определение 43.

$$x \in \Upsilon(t, l, r) \Leftrightarrow (x == t) \vee (x \in l) \vee (x \in r)$$

Определение 44. ***Дерево поиска ИЛИ упорядоченное, слева направо,*** —

- *пустое дерево* $\varnothing$
- *структурный терм* $\Upsilon(x, l, r)$
  - Υ – *имя терма.*
  - x – *узел.*
  - l – *левое поддерево.* $\forall \xi \in l \xi < x$
  - r – *правое поддерево.* $\forall \xi \in r \xi > x$

III. Вопрос 25.2

Операции:

- `in( X, S )`. Проверка принадлежности элемента X к набору данных S.

```
1       in(X,t(X,_,_)).
2       in(X, t(_,L,_) ) :-
3         in( X, L) .
4       in( X, t(_,_,R):-
5         in ( X, R) .
```

  Элемент X находится в дереве поиска.

```
1       in( X, t ( X, _, _) ).
2       in( X, t( Root, Left, Right) ) :-
3         gt(Root, X), % Корень больше, чем X
4         in( X, Left}. % Проводить поиск в левом поддереве
5       in( X, t( Root, Left, Right} ) :-
6         gt( X, Root}, % X больше, чем корень
7         in( X, Right). % Проводить поиск в правом поддереве
```

  Применение:

  – Поиск

  – Построение

- `add( S, X, S1 )`. Добавление элемента X. к набору данных S и получение набора данных S1.

```
1       addleaf( nil, X, t(X, nil, nil)).
2       addleaf( t(X, Left, Right), X, t(X, Left, Right)).
3       addleaf( t(Root, Left, Right), X, t(Root, Left1, Right)):-
4         gt(Root, X),
5         addleaf( Left, X, Left1).
6       addleaf( t(Root, Left, Right), X, t(Root, Left, Right1)):-
7         gt(X, Root),
8         addleaf( Right, X, Right1).
```

- `del( S, X, S1)`. Удаление элемента X из набора данных S и получение набора данных S1.

```
1       del( t(X,nil, nil), X, nil ).
2       del( t(X,Left, nil), X, Left ).
3       del( t(X,nil, Right), X, Right ).
4       del( t(X,Left, Right), X, t(Y,Left, Right) ):−
5         delmin( Right, Y, Right1 ).
6       del( t(Root,Left, Right), X, t(Root,Left1, Right) ):−
7         gt(Root, X),
8         del( Left, Y, Left1 ).
9       del( t(Root,Left, Right), X, t(Root,Left, Right1) ):−
10        gt(X, Root),
11        del( Right, Y, Right1 ).
12      delmin(t(Y, nil, R), Y, R).
13      delmin(t(Root,Left, Right), Y, t(Root,Left1, Right)):−
14        delmin(Left, Y, Left1).
```

Вставка и удаление, на любом уровне, в одном лице: Хотим ввести X в дерево D

- Ввод X в корень. X – новый корень
- Перемешения:
  - $D > X$ вставить в левое поддерево
  - $D < X$ вставить в правое поддерево

Страница 203/205 Ивана Братко.

```
1   add(Tree, X, NewTree):− addroot(Tree, X, NewTree).
2   add(t(Y, L, R), X, t(Y, L1, R)):− gt(Y, X), add(L, X, L1).
3   add(t(Y, L, R), X, t(Y, L, R1)):− gt(X, Y), add(R, X, R1).
4   addroot(nil, X, t(X, nil, nil)).
5   addroot(t(Y, L, R), X, t(X, L1, t(Y, L2, R))):−
6     gt(Y, X), addroot(L, X, t(X, L1, L2)).
7   addroot(t(Y, L, R), X, t(X, t(Y, L, R1), R2)):−
8     gt(Y, X), addroot(R, X, t(X, R1, R2)).
```

26 Сбалансированные деревья

Сбалансированные деревья. Алгоритм добавления узла в сбалансированное дерево.

Дерево называется (примерно) сбалансированным, если для каждого узла в дереве два поддерева этого узла содержат приблизительно равное количество элементов.
Примерно сбалансированные деревья:

- AVL-дерево
- Двоично-троичное дерево.

$<?>$ **1.** *Какое из них подразумевается в вопросах*

Определение 45. ***Дерево AVL*** *—*

- *пустое дерево* $\varnothing$
- *структурный терм* $\Upsilon(t, l, r)$
  - Υ *– имя терма.*
  - t *– произвольный терм (узел).*
  - l *– левое AVL-поддерево.*
  - r *– правое AVL-поддерево.*

  и $|h(l) - h(r)| \leqslant 1$*, где* $h(\cdot)$ *глубина*

Определение 46. $h(\cdot)$ *— глубина дерева*

- *пустое дерево* $h(\varnothing) = 0$
- $h(\Upsilon(t, l, r)) = \max(h(l), h(r)) + 1$

AVL-деревья представляют `t(Root, Left, Right)/Height`

$<?>$ **2.** *Что вобщее требуется тут?*

III. Вопрос 26.2

Комментарий Дениса Долгова:

Сбалансированные деревья нужны для быстрого поиска, по крайне мере такое назначение им давалось в книге Сошникова. Потому как просто двоичные деревья поиска, могут не обладать всей привлекательностью двоичного поиска. Так например строя обычное двоичное дерево поиска для отсортированной последовательности, оно получится не как дерево поиска, а как линейный список, а как известно свободного доступа к элементам списка в Прологе нет. Поэтому в общей сложности поиск в таком дереве составит $O(n)$, то есть линейный. Для того, чтобы сохранять свойства двоичного поиска, а точнее сложность $O(log2(n))$, можно использовать сбалансированные деревья. На сколько помню сбалансированным деревом называется двоичное дерево, высота правого и левого поддерева которого по разности не превосходит единицу, а также эти правое и левое поддерево, также являются сбалансированными. Поэтому даже строя сбалансированное дерево для упорядоченной последовательности, не теряется быстрота поиска. Но тут добавление в дерево связано с трудностями, потому как может нарушится сбалансированность. Поэтому дерево иногда приходится перестраивать. Код достаточно большой для этой операции, и думаю написать его полностью на экзамене, чётко не представляя в голове тяжело. Поэтому думаю хватит основных идей как осуществляется вставка в сбалансированное дерево. Тем более на экзамене и так есть задачки по Прологу.

<?> 3. *Нужен ли код?*

27 Представления графов

Представления графов. Алгоритмы поиска пути в графе (в глубину, в ширину).

Определение 47. ***Направленый граф*** $\langle U, V \rangle$ — *пара*

- U *вершины*
- $V \subseteq U \times U$ *ребра (дуги для направленного)*

Для представления графов обычно используют матрицы смежности или динамические списочные структуры. В логическом программировании удобнее представлять граф в виде пар дуг. Это очень похоже на использование матрицы смежности, но удобнее, естественнее и экономичнее. + в логическом программировании мы можем задавать дуги не только фактами, но и правилами.

Определение 48. ***Путь в графе*** $G =< U, V >$ *из* $a \in U$ *в* $b \in U$ — *последовательность* $\{x_i\}_{i=0}^{n}$, *такую, что*

$$\begin{aligned} x_i &\in U, \\ x_0 &= a \\ x_n &= b \\ \forall i \langle X_{i-1}, x_i \rangle &\in V \end{aligned}$$

```
1  % Граф
2  door(a, b).
3    door(b, c).  door(b, d).
4      door(c, d).
5        door(d, e).  door(d, f).
6          door(f, g).
7            door(h, i).
8  % Движение по графу
9  move(X, Y):- door(X, Y), door(Y, X).
```

27.1 В глубину

Для исследования пути l_{max} и дерева с коэффициентом ветвление b:
$TIME = O(l_{max})$
$SPACE = O(b^{l_{max}})$
В процессе поиска путь из начальной вершины продлевается одним из возможных способов, пока не будет продлён в конечную вершину. Углубляется по дереву. Альтернативный путь – только в случае неудачи.

```
search_depth(A,B,P):-
  depth([A], B, L), reverse(L, P).
depth([X|T], X, [X|T]).
depth(P, F, L):-
  prolong(P, P1), depth(P1, F, L).
prolong([X| T], [Y, X |T])
  move(X, Y), not(member(Y, [X | T])).
```

27.2 В ширину

Для исследования некоторого пути l и дерева с коэффициентом ветвление b:
$TIME = O(b^l)$
$SPACE = O(b^l)$
Рассматриваем очередь путей, содержащие путь из начальной вершины. На каждом шаге

1. берем путь из начало очереди,
2. продляем его (всеми способами) и вставляем в конец.
3. Если путь в начале очереди – решение, – *ПРИНЯТЬ*

Не виснет. Ищет кратчайшее решение раньше остальных. Идеи сходны с алгоритмом фронта волны но матрицы смежности заменены продлением списков. Это позволяет проводить вычисление на графах, где матрицы смежности уже неприменимы, из-за своего объёма.

```
search_bdth(X, Y, P):-
  bdth([[X]], Y, P), reverse(P, L).

bdth([[X|T]|_], X, [X|T]).
bdth([P|QI], X, R):-
  findall(Z, prolong(P, Z), T),
  append(QI, T, QO),!,
  %% append(T, QI, QO)
  /*
    ставим это вместо пардыдущей строки ==> В глубину
    очередь путей в стек.
  */
  bdth(QO, X, R).
bdth([_|T], Y, L):-
  bdth(T, Y, L).
```

IV Методы решения задач

28 Метод генерации и проверок для решения задач

Метод генерации и проверок для решения задач. Метод ветвей и границ. Явный и неявный перебор в Пролог-программах.

28.1 Метод генерации и проверок

- Один предикат генерирует множество исходных данных
- Второй проверяет на соответствие с условиями задачи.
  - Успех – возвращаем.
  - НеУспех – backtracking.

На самом деле любое доказательство так или иначе использует этот метод При этом промежуточные этапы проверки отсекают дальнейшие ветки от полного дерева перебора, реализуя метод ветвей и границ

28.2 Метод ветвей и границ

Части общего дерева решения отсекаются на ранних этапах или вообще не генерируются То, насколько много веток отсекается, определяет эффективность решения задачи

Важно

- Выбрать удобное представление состояния задачи
- Построить перебор методов ветвей и границ, чтобы отсекать наиболее объёмные ветки дерева
- Иногда неявный перебор и правильный выбор способа представления знаний позволяет решить задачи в чисто декларативном стиле

При использовании **явного перебора** — производится полная конкретизация решений — а потом ух проверка.

Неявный перебор использует метод ветвей и границ — генерация очередной порции решений производится после частичной конкретизации решения.

<?> **4.** *Нужен ли тут код вообще ?*

29 Основные методы сокращения перебора на примере задач криптографии, расстановки ферзей, игры $N^2 - 1$

Основные методы сокращения перебора на примере задач криптографии, расстановки ферзей, игры $N^2 - 1$.

$<?>$ **5.** *Непонятно! — криптография*

IV. Вопрос 29.2

На шахматной доске размером $N \times N$ требуется расставить N ферзей, так, чтобы они не били друг друга.

Если представить позицию в виде набора из координат x и y всех ферзей, то задачу можно сформулировать как задачу на удовлетворение ограничений.

Найти набор значений для $x_1, y_1, x_2, y_2, ..., x_N, y_N$, из множества $[1..N]$ такой, что для всех различных i и j выполняются условия:

- $x_i \neq x_j$ (ферзи не стоят на одной вертикали);
- $y_i \neq y_j$ (ферзи не стоят на одной горизонтали);
- $|x_i - x_j| \neq |y_i - y_j|$ (ферзи не стоят на одной диагонали). Можно попытаться решить эту задачку методом перебора

- Плохой вариант

```
1 queens(N,Ys−Xs) :−
2     range(1,N, Ns),
3     length(Xs, N), members(Xs,Ns),
4     length(Ys, N), members(Ys,Ns),
5     safe(Ys−Xs).
```

Однако размер пространства перебора $N^{2 \cdot N}$ должен насторожить. Для классического случая $N = 8$ придется рассмотреть порядка 10^{14} вариантов. Заметим, что среди координат x ферзей $x_1, x_2, ..., x_N$ ровно один раз встречается каждое из чисел $1, 2, ..., N$. Таким образом, номера ферзей можно рассматривать как координаты x и достаточно перебирать только координаты y. Новая формулировка задачи:

Найти набор (y1,y2,...,yN), 1<yi<N такой, что для всех различных i и j выполняются условия:

yi =/= yj |yi - yj| =/= | i-j |.

Новая схема перебора будет выглядеть так

```
1 queens(N,Ys) :−
2     range(1,N, Ns),
3     length(Ys, N),
4     members(Ys, Ns),
5     safe(Ys).
```

Теперь пространство перебора содержит NN (для N=8 -107) вариантов, что уже лучше, но все еще многовато. Если учесть, что среди координат y ферзей y1,y2,...,yN

каждое из чисел 1,2,...,N также встречается ровно один раз, можно еще раз переформулировать задачу.

Найти набор различных (y1,y2,...,yn), 1<yi<N, такой что |yi - yj| =/= | i-j | при i =/= j.

Конечно это, по сути, то же самое, просто мы переносим условие yi =/= yj из множества ограничений в определение пространства перебора. Процедура перебора изменяется незначительно.

```
1  queens(N,Ys) :−
2      range(1,N, Ns),
3      length(Ys, N),
4      selects(Ys, Ns),
5      safe(Ys).
```

Пространство перебора сокращается с NN до N!, что для N=8 означает 40 320 и уже приемлемо. Теперь можно записать процедуру проверки.

```
1  safe([Y|Ys]) :− noattack(Y,Ys), safe(Ys).
2  safe([]).
3  noattack(Y,Ys) :− noattack(Y,Ys,1).
4  noattack(_,[],_).
5  noattack(Q,[Y|Ys],D) :−
6      abs(Q−Y) =\= D,
7      D1 is D+1,
8      noattack(Q,Ys,D1).
```

Однако факториал есть факториал и если для N=8 время поиска всех ответов измеряется секундами, то для N=10 это уже минуты, а для N=12 - часы. Ничего поделать с природой задачи мы не можем - невозможно сделать из свиньи скаковую лошадь. Но можно получить более быструю свинью. Мы можем, как и в предыдущем примере, переместить проверку ограничений в процедуру генерации. Выбирая положение очередного ферзя, будем проверять его совместимость с расставленными ранее. Тем самым мы объединим процедуры selects и safe в одну процедуру `place_queens`. Эта процедура имеет дополнительный аргумент Board, содержащий список уже расставленных ферзей.

```
1  queens(N,Ys) :−
2     range(1,N,Ns),
3     length(Ys,N),
4     place_queens(Ys,Ns,[]).
5  place_queens([Y|Ys], Ns, Board) :−
6      select(Y, Ns, Ns1),
7      noattack(Y, Board),
```

```
8       place_queens(Ys, Ns1, [Y|Board]).
9   place_queens([],_,_).
```

В сравнении с предыдущей эта программа выполняется раз в 20 быстрее при N=8, а кроме того время вычислений пропорционально уже N!0.7.

Заметим, что процедура `place_queens` сама содержит пару из генератора и теста. Мы сначала выбираем положение ферзя посредством select, а затем проверяем его на совместимость, вызывая noattack. Заманчиво объединить и эти две процедуры. Однако, чтобы такое объединение имело смысл, необходимо научиться выбирать только допустимые положения. Для этого потребуется нечто большее, чем просто список уже расставленных ферзей. В своей знаменитой статье [1] Никлаус Вирт использовал два булевых массива представляющих /-диагонали и диагонали. Мы несколько модифицируем эту идею, и будем использовать два списка Us и Ds, содержащие переменные, "охраняющие"диагонали. Каждая переменная представляет одну диагональ. При размещении очередного ферзя переменная из списка Ys связываетя с номером ферзя, но одновременно мы будем связывать соответствующие переменные из списков Us и Ds. Поскольку переменная может только иметь одно значение, как только она связана, никакой другой ферзь уже не может оказаться на той же диагонали (этот же трюк мы применяли при раскраске карты). Процедура `place_queens` после выбора положения для ферзя "сдвигает"Us и Ds на одну позицию в разные стороны.

```
1   queens(N,Ys) :−
2       range(1,N, Ns),
3       length(Ys, N),
4       place_queens(Ns,Ys,_,_).
5   place_queens([N|Ns], Ys, Us, [D|Ds]):−
6       place1(N, Ys, Us, [D|Ds]),
7       place_queens(Ns, Ys, [U|Us], Ds).
8   place_queens([], _, _, _).
9   place1(N,[N|_], [N|_],[N|_]).
10  place1(N,[_|Cs],[_|Us], [_|Ds]):− place1(N,Cs,Us,Ds).
```

Такое усовершенствование дает выигрыш в скорости в 2-3 раза, но в целом оказывается не очень существенным. Пространство перебора остается, по сути, тем же самым. Ускоряется лишь проверка возможности очередного выбора. Тем не менее, эта программа иллюстрирует важный метод решения задач, называемый распространением ограничений (constraints propagation). Каждый выбор значения для очередной переменной ограничивает возможность выбора для других переменных.

Распространение ограничений - довольно общий принцип, допускающий разнообразные реализации. Важный способ реализации этого метода - вести явный учет возможных значений для каждой переменной, в нашем случае - возможных положений

для каждого ферзя. Этот метод называется "просмотром вперед"(forward checking). Заведем список пар (номер ферзя : список допустимых положений). Вначале каждый ферзь получает полную свободу, его список имеет вид [1,2,...N]. Размещая очередного ферзя, мы удаляем атакованные им позиции из допустимых положений оставшихся ферзей, ограничивая тем самым возможность выбора.

```
1  queens(N, Queens) :−
2      range(1,N, Ns),
3      init_vars(Ns, Ns, V),
4      place_queens(V,Queens).
5  init_vars([X|Xs],Ys,[X:Ys|V]) :−
6      init_vars(Xs,Ys,V).
7  init_vars([], _, []).
8  place_queens([X:Ys|V],[X−Y|Qs]) :−
9      member(Y, Ys),
10     prune(V, X, Y, V1),
11     place_queens(V1,Qs).
12 place_queens([],[]).
13 prune([Q:Ys|Qs], X,Y, [Q:Ys1|Ps]) :−
14     sublist(noattacks(X,Y,Q), Ys, Ys1),
15     Ys1 \== [],
16     prune(Qs, X,Y, Ps).
17 prune([], _,_, []).
18 noattacks(X1,Y1,X2,Y2) :−
19     Y1 \== Y2,
20     abs(Y2 − Y1) =\= abs(X2 − X1).
```

Встроенный предикат sublist - аналог функции filter. Он возвращает список тех элементов списка, для которых выполняется указанный предикат. В сущности, это более быстрый способ выполнить

```
1  findall(T, (member(T,Ys),noattacks(X,Y,Q,T)) , Ys1)
```

Обратите внимание на проверку Ys1 $\doteq$ [] . Если для какого либо из еще не размещенных ферзей не остается свободных мест, prune завершается неудачей. Таким образом, предполагаемое положение ферзя отвергается сразу.

Теперь предположим, что мы не стали включать указанную проверку в prune. Тогда, вместо того чтобы тупо расставлять ферзей одного за другим, разумно сначала убедиться, что среди не размещенных ферзей нет таких, которые нельзя разместить. Далее надо посмотреть, нет ли таких ферзей, для которых существует всего один вариант размещения и, если есть, выбрать именно такого. Аналогично, в общем случае лучше выбирать того ферзя, для которого осталось меньше всего свободных

мест. Итак, мы можем усовершенствовать программу, изменив порядок расстановки ферзей. Поскольку теперь ферзи необязательно будут размещаться по порядку, результат будем представлять в виде списка пар X-Y.

```
1 place_queens(V, [X−Y|Qs]) :−
2     queen_to_place(V,X:Ys,V1),
3     member(Y, Ys),
4     prune(V1, X, Y, V2),
5     place_queens(V2,Qs).
6 place_queens([], []).
```

Процедура `queen_to_place` выбирает ферзя подлежащего размещению из списка не размещенных. Если запишем просто

```
1 queen_to_place([V|Vs],V,Vs).
```

то получим предыдущий вариант. Мы же будем выбирать "наиболее ограниченного"ферзя, то есть того, у которого список возможных положений самый короткий.

```
1 queen_to_place([V|Vs], Q, Vx) :− select_boundest(Vs, V, Q, Vx).
2 select_boundest([], Q, Q, []).
3 select_boundest([X|Xs], Q, Qm, [Q|R]) :−
4     shorter(X, Q), !,
5     select_boundest(Xs, X, Qm, R).
6 select_boundest([X|Xs], Q, Qm, [X|R]) :−
7     select_boundest(Xs, Q, Qm, R).
8 shorter(_:L1, _:L2) :− length(L1,N1), length(L2,N2), N1<N2.
```

При сокращении пространства перебора надо соотносить размер этого сокращения с затратами на его реализацию. На небольших досках эта программа оказывается даже медленнее - затраты на реализацию более сложной схемы съедают весь выигрыш от сокращения. Тем не менее, с ростом N сокращение играет все большую роль. Время, выполнения программы растет как квадратный корень из N!, а это означает, что мы сможем решить задачу для досок большего размера.

30 Решение задач методом поиска в пространстве состояний

Решение задач методом поиска в пространстве состояний. Пример. Принципы подбора алгоритма поиска пути в зависимости от задачи.

IV. Вопрос 31.1

31 Операторы

Операторы. Преобразование символьных выражений в дерево. Вычисление значения арифметического выражения по строке.

32 Символьные вычисления

Символьные вычисления. Алгоритм символьного дифференцирования.

33 Подходы к символьному упрощению выражений

Подходы к символьному упрощению выражений.

34 Анализ естественного языка с использованием контекстно-свободной грамматики

Анализ естественного языка с использованием контекстно-свободной грамматики. Подход к интерпретации знаний в естественно-языковом материале.

35 Учёт контекстных условий

Учёт контекстных условий. Работа со словарями. Глубинные и поверхностные структуры в естественном языке.

36 Расширение логического интерпретатора Пролога средствами языка

Расширение логического интерпретатора Пролога средствами языка. Мета-интерпретаторы. Примеры.

V Приложение

1 Общие вопросы

<?> **1.** *Насколько надо знать Mercury ?*

<?> **2.** *Сколько кода приводить в ответах на теорию ?*

<?> **3.** *Можно своими словами или с формулами ?*

2 Статистика

Тестирование $\rho \gamma \LaTeX$: $\Delta \alpha$

```
1  Тестирование $\rho \gamma \LaTeX$ :
2  \begin{python}
3    print "$\\Delta \\alpha$ \n\n"
4  \end{python}
```

Время последней компиляции: Tue Jun 30 11:51:32 2009

Операционная система: win32 nt

3 Итого.Параметры подготовки

Вопросов 8 штук

Определений 48 штук