1 От математической логики к логическим программам

1.1 Язык логики первого порядка и конструктивные задачи

Каждый язык, заполняющий определенную нишу в сфере программистских задач, имеет свою метафору, которая сжато и приблизительно определяет его назначение. Например, метафора языка ассемблера – "язык абстрактной вычислительной машины фон Неймана", метафора Си – "структурная оболочка языка ассемблера, ориентированная на программирование функций", метафора Ады – "язык модульного программирования и проектирования встроенных систем". Языки логического программирования (ЯЛП) вводят новую метафору. Первый член этой метафоры – "Языки, в которых решение задачи в символическом виде заменяется формулировкой ее условия и вопроса на языке логики предикатов [1]". Эта часть метафоры уподобляет языки логического программирования языкам запросов реляционых баз данных (РБД), но не позволяет различить их между собой. Существует два основных отличия ЯЛП от языков запросов РБД. Во-первых, их интерпретатор рекурсивен, тогда как в языках РБД рекурсия невыразима. Во-вторых, интерпретатор языка запросов РБД неизвестен пользователю, и потому у него нет способа управлять поиском решения, в то время как интерпретатор ЯЛП известен и столь прост, что, имея его в виду, можно программировать процесс поиска решения вопроса, выбирая подходящую формулировку условия задачи. Таким образом, второй член метафоры ЯЛП таков: "программирование решения задачи – это выбор некоторой ее формулировки, характеризующей определенный способ решения".

(1) Языки, в которых решение задачи в символическом виде заменяется формулировкой ее условия и вопроса на языке логики предикатов. (2) Программирование решения задачи – это выбор некоторой ее формулировки, характеризующей определенный способ решения.

.

Язык логических программ – это некоторая разновидность языка логики предикатов первого порядка $\mathcal{L}$. Логическая программа – это некоторая конечная теория $\mathcal{P}$ в языке $\mathcal{L}$ (условия задачи) и формула этого языка ϕ (вопрос). Условие ее разрешимости означает существование такой подстановки θ значений переменных в ϕ, что результат подстановки $\phi \circ \theta$ является следствием $\mathcal{P}$:

$$\mathcal{P} \Longrightarrow \phi \circ \theta.$$

Особенность ЯЛП – это наличие конструктивной интерпретации следования. Для логической программы это означает, что искомая подстановка θ строится алгоритмом по $\mathcal{P}$ и ϕ. Таким образом, логические программы предназначены для решения конструктивных вычислительных задач существования объектов. Приведем простой пример формулировки вычислительной задачи в форме логической программы.

Пример 1.
Содержательная формулировка условия задачи: *Каждый следующий член последовательности натуральных чисел на 1 меньше предыдущего.*

[1] Мы предполагаем знакомство с основными понятиями математической логики в объеме стандартного университетского курса, соответствующего, например, первым главам книг [1, 2]

Содержательная постановка вопросов:

ВОПРОС 1: Найти последовательность, начинающуюся с 6,X,Y.

ВОПРОС 2: Найти последовательность вида 6,X,1.

Определим сигнатуру нашего языка. Прежде всего, договоримся, что последовательности будут представляться выражениями - списками.

Определение 1. *Пусть* **Var** *- счетное множество предметных переменных, и* **O** *– некоторое множество объектов. Тогда множество* $\mathbf{lst}(\mathbf{O},\mathbf{Var})$ списков *над* $\mathbf{O},\mathbf{Var}$ *определяется так:*
$[\,] \in \mathbf{lst}(\mathbf{O},\mathbf{Var})$ *(пустой список).*
Если $l_1,...,l_n \in \mathbf{O} \cup \mathbf{Var} \cup \mathbf{lst}(\mathbf{O},\mathbf{Var})$ *и* $L \in \mathbf{Var} \cup \mathbf{lst}(\mathbf{O},\mathbf{Var})$, *то выражения* $[l_1,...,l_n|L]$ *также принадлежит* $\mathbf{lst}(\mathbf{O},\mathbf{Var})$.
При этом предполагается, что при $n > 1$
(L_1) $\quad [l_1,...,l_n|L] = [l_1,...,l_{n-1}|[l_n|L]]$
и при $n = 1$
(L_2) $\quad [l_1] = [l_1|[\,]]$.

Выражение $[\,]$ задает пустой список. Выражение $[E|X]$ задает список с *первым элементом* E и *хвостом* X. Т.о., например, $[1,2] = [1|[2]] = [1|[2|[\,]]]$. Списки, у которых хвост является переменной, называются *недоопределенными*. Например, если X,T – переменные, то $[a,[\,],X|T]$ задает список, начинающийся элементами $a,[\,],X$. Заметим, что, выделив в сигнатуре константу $nil/0$ и двухместный функтор $"."/2$, можно было бы описывать списки обычными префиксными термами. Например, пустой список $[\,]$ соответствовал бы терму nil, выражение $[1,2]$ - терму $.(1,.(2,nil))$, а выражение $[1,2|X]$ - терму $.(1,.(2,X))$. Мы однако будем использовать принятое в логическом программировании более удобное представление списков, и при этом обращаться с ними как с обычными термами.

Вернемся к задаче из примера 1. Теперь, чтобы сформулировать ее условие на языке логики предикатов, следует ввести одноместный предикат mds, обозначающий свойство списка "быть монотонно убывающей последовательностью натуральных чисел с шагом 1", и определить это свойство аксиоматически. Для простоты будем считать, что наша логика является прикладной в том смысле, что в ее сигнатуре уже присутствуют константы для натуральных чисел, арифметические операторы и предикаты, термы для списков, и в качестве моделей рассматриваются обогащения стандартной арифметической структуры, удовлетворяющие тождествам L_1, L_2. Тогда аксиомы, описывающие условия задачи для mds, можно записать так:

Логическая программа:

(a_1) $\quad mds([\,])$.

(a_2) $\quad \forall X \bullet mds([X])$.

(a_3) $\quad \forall E,F,T \bullet (mds([E|T]) \;\&\; F = E+1 \longrightarrow mds([F,E|T]))$.

Легко записываются и оба вопроса:

ВОПРОС 1: $\exists X,Y,T \bullet mds([6,X,Y|T])$.

ВОПРОС 2: $\exists X \bullet mds([6,X,1])$.

Таким образом, мы сформулировали две вычислительные задачи:

$\{\mathsf{a}_1, \mathsf{a}_2, \mathsf{a}_3\} \Longrightarrow \exists X, Y, T \bullet mds([6, X, Y|T])$ и

$\{\mathsf{a}_1, \mathsf{a}_2, \mathsf{a}_3\} \Longrightarrow \exists X \bullet mds([6, X, 1])$.

Первую задачу корректный интерпретатор логических программ должен решить положительно, и дать, например, ответ: $X = 5, Y = 4, T = [\,]$, а вторую он должен решить отрицательно.

1.2 Полнота языка простых дизъюнктов в эрбрановских интерпретациях [2]

Мы рассматриваем язык логики предикатов первого порядка в некоторой сигнатуре предикатных символов $\mathbf{P}$ и символов функторов $\mathbf{F}$. Договоримся для тех и других выбирать имена, начинающиеся с прописных букв, и, если требуется уточнение, записывать символ a арности $n \geq 0$ в виде a/n. Константы – это нульместные функторы: $f/0$. Считается, что в сигнатуре присутствует счетное множество констант. Мы будем предполагать наличие счетного множества предметных переменных $\mathbf{Var}$ и договоримся выбирать для переменных имена, начинающиеся с заглавных букв. Под *теорией* будет пониматься любое множество замкнутых формул в данной сигнатуре.

Как обычно, истинность формулы ϕ в сигнатурной интерпретации I при означивании предметных переменных σ будет обозначаться через $I, \sigma \models \phi$. Соответственно, $I, \sigma \models \mathcal{P}$ будет означать,что $I, \sigma \models \phi$ имеет место для всех формул $\phi \in \mathcal{P}$. Формула ϕ *истинна при интерпретации* I, или I *является моделью* ϕ (обозначение $I \models \phi$), если при любом означивании переменных σ имеет место $I, \sigma \models \phi$. Поэтому, например, для любой замкнутой формулы всякая интерпретация является либо ее моделью, либо моделью ее отрицания. Множество формул *совместно*, если оно имеет модель, и *несовместно* в противном случае. Напомним, что формула ϕ *[логически] следует* из множества формул $\mathcal{P}$ (обозначение: $\mathcal{P} \Longrightarrow \phi$), если для любых I и σ $I, \sigma \models \mathcal{P}$ влечет $I, \sigma \models \phi$. Более слабое отношение $\mathcal{P} \models \phi$ означает, что всякая модель $\mathcal{P}$ является моделью ϕ. Более общо: $\mathcal{P}_1 \Longrightarrow \mathcal{P}_2$ $(\mathcal{P}_1 \models \mathcal{P}_2)$ означает, что для всякой формулы $\phi \in \mathcal{P}_2$ имеет место $\mathcal{P}_1 \Longrightarrow \phi$ (соответственно $\mathcal{P}_1 \models \phi$). Нам будет полезно различать два отношения эквивалентности на множествах формул: *равносильность* и *равносовместность*.

Множества формул $\mathcal{P}_1$ и $\mathcal{P}_2$

- *равносильны* (обозначение: $\mathcal{P}_1 \equiv \mathcal{P}_2$), если $\mathcal{P}_1 \models \mathcal{P}_2$ и $\mathcal{P}_2 \models \mathcal{P}_1$,
- *равносовместны* (обозначение: $\mathcal{P}_1 \equiv_m \mathcal{P}_2$), если $\mathcal{P}_1$ совместна т. и т.т., когда совместна $\mathcal{P}_2$.

Понятно, что для любой системы формул Γ и для любой формулы ϕ $\Gamma \Longrightarrow \phi$ влечет несовместность системы формул $\Gamma' = \Gamma \cup \{\neg\phi\}$. Обратное, вообще говоря, неверно. Например, система формул $\{p(X), \neg\forall X \bullet p(X)\}$ несовместна, однако, неверно $p(X) \Longrightarrow \forall X \bullet p(X)$. Между тем, имеет место

Предложение 1. *Для всякой теории $\mathcal{P}$ и всякой замкнутой формулы ϕ несовместность $\mathcal{P} \cup \{\neg\phi\}$ влечет $\mathcal{P} \Longrightarrow \phi$.*

[2] Материал этого раздела отчасти покрывается главой 4 книги [4] и главой 1 книги [5].

Следует заметить, что в условиях этого предложения $\mathcal{P} \Longrightarrow \phi$ равносильно $\mathcal{P} \models \phi$.

Вернемся к общей вычислительной задаче

$$\mathcal{P} \Longrightarrow \exists \bar{X} \bullet \phi.$$

Сформулированное предложение говорит, что эта задача разрешима т. и т.т., когда система формул $\mathcal{P} \cup \{\forall \bar{X} \bullet \neg\phi\}$ несовместна. Таким образом, разрешимость вычислительной задачи сводится к несовместности некоторой конечной теории. Это сведение позволяет произвести кардинальные упрощения в языке формулировки вычислительных задач, не теряя общности.

Назовем ϕ формулой в *литеральной форме*, если в ϕ не встречаются связки $\longrightarrow$ и $\longleftrightarrow$, а $\neg$ относится лишь к атомарным подформулам. Как известно, любая формула может быть приведена к литеральной форме эквивалентными преобразованиями. Кроме того, если $\bar{X}$ – все свободные переменные ϕ, то формула $\forall \bar{X} \bullet \phi$ очевидно равносовместна с формулой ϕ ($\phi \ \equiv_m \ \forall \bullet \bar{X}\phi$). Основным инструментом упрощения является следующая лемма.

Пусть $\phi = \phi[\exists X \bullet \psi]$ – замкнутая формула в литеральной форме, в которой выделенное вхождение экзистенциальной подформулы является самым левым. Это, в частности, означает, что все переменные $\bar{Y}$, являющиеся в ψ свободными, связаны в ϕ квантором $\forall$. Пусть r – число переменных в $\bar{Y}$ $(r \geq 0)$ и f/r – некоторый новый (т.е. не встречающийся в ϕ) функтор из $\mathbf{F}$. Таким образом, при $r = 0$ f является новой константой. Преобразуем ϕ, заменяя в ней выделенное вхождение $\exists X \bullet \psi$ на ψ и каждое свободное вхождение X в ψ на терм $t = f(\bar{Y})$, получая таким образом новую формулу $\phi^s = \phi[\psi \mid {}^{t}_{X}]$.

Лемма 1 (Сколем).
$\phi \ \equiv_m \ \phi^s$ (ϕ имеет модель т. и т.т., когда ее имеет ϕ^s).

Лемма Сколема позволяет (с использованием эквивалентных преобразований) преобразовать любую конечную теорию $\mathcal{P}$ в равносовместную универсальную формулу в предваренной нормальной форме вида $\forall \bar{X} \bullet \psi$. Таким образом, из этой леммы непосредственно вытекает

Следствие 1. *Для любой конечной теории имеется равносовместная универсальная формула, матрица которой является формулой в конъюнктивной нормальной форме.*

Назовем *литералом* атомарную формулу или ее отрицание, *простым дизъюнктом* – формулу, являющуюся универсально замкнутой дизъюнкцией литералов $\forall \bar{X} \bullet (l_1 \vee ... \vee l_n)$, и *простым предложением* – формулу одного из следующих трех типов:

$\forall \bar{X} \bullet (a_1 \vee ... \vee a_n)$, где a_i – атомарные формулы (*факт*),

$\forall \bar{X} \bullet \neg(a_1 \& ... \& a_n)$, где a_i – атомарные формулы (*цель*),

$\forall \bar{X} \bullet (a_1 \& ... \& a_n \longrightarrow h_1 \vee ... \vee h_m)$, где a_i, h_j – атомарные формулы (*простая импликация*).

Из следствия 1, в свою очередь, вытекает

Следствие 2. *Для любой конечной теории имеется равносовместная конечная теория, состоящая только из простых дизъюнктов или, соответственно, только из простых предложений.*

Очевидно, что сведение, описываемое в сформулированных предложениях, конструктивно. Оно реализуется некоторым алгоритмом, который называют *сколемизацией*. Поскольку результатом сведения всегда является универсальная теория, мы будем опускать кванторные приставки формул.

Пример 2. *Рассмотрим теорию* $\mathcal{P} = \{\phi_1, \phi_2\}$, *где*
ϕ_1 : $\neg\exists X \bullet s(X)\ \&\ \neg c(X)$
ϕ_2 : $\forall X \bullet (c(X)\ \vee\ \forall Z \bullet v(X,Z))$,
и формулу
ϕ : $\forall X, Y \bullet (s(X) \longrightarrow v(X,Y))$.
Чтобы показать, что $\mathcal{P} \Longrightarrow \phi$, *рассмотрим систему формул* $\mathcal{P}' = \mathcal{P} \cup \{\neg\phi\} = \{\phi_1, \phi_2, \exists X, Y \bullet (s(X)\ \&\ \neg v(X,Y))\}$. *В результате сколемизации эта система преобразуется в равносовместную систему простых предложений* Γ:

$\neg c(X) \longrightarrow v(X,Y)$,
$s(Z) \longrightarrow c(Z)$,
$s(c_1)$,
$\neg v(c_1, c_2)$,

которая, очевидно, несовместна.

Сделаем важное замечание. Благодаря предложению 1, сколемизация позволяет исследовать разрешимость вычислительной задачи. Но она не дает возможности описывать решения в случае, когда формула-вопрос оказывается следствием формул-условий задачи. Дело в том, что мы оперируем произвольными интерпретациями теорий, и таким образом, не располагаем единым универсумом, над которым можно было бы определять подстановки-решения. Однако, как мы убедились, разрешимость вычислительной задачи сводится к совместности системы *универсальных* формул. А для таких систем оказывается достаточно рассматривать интерпретации с общим универсальным множеством. Эта идея принадлежит Эрбрану.

Определение 2. *Пусть* $\mathcal{P}$ *– некоторая универсальная теория в сигнатуре* $\mathbf{P}, \mathbf{F}$ *и* $\mathbf{H}_\mathcal{P}$ *– множество всех* базисных *(т.е. не содержащих переменных) термов над* $\mathbf{F}$ *–* эрбрановский универсум $\mathcal{P}$. Эрбрановский базис $\mathcal{P}$ *– это множество* $\mathbf{B}_\mathcal{P}$ *всех атомов вида* $p(t_1, ..., t_k)$, *в которых* $t_i \in \mathbf{H}_\mathcal{P}, 1 \le i \le k$. Эрбрановской *называется всякая интерпретация* I *с универсумом* $\mathbf{H}_\mathcal{P}$, *в которой для любого функтора* f/r $(r \ge 0)$ *и любых базисных термов* $t_1, ..., t_r \in \mathbf{H}_\mathcal{P}$ *имеет место*

$$f(t_1, ..., t_r)^I = f(t_1^I, ..., t_r^I).$$

Т.о., эрбрановские интерпретации задают символические вычисления. Например, в эрбрановской интерпретации значением терма $(5+1)^2$ является сам этот терм, а не число 36. Поэтому эрбрановские интерпретации удобно представлять попросту как подмножества эрбрановского базиса: для $I \subseteq \mathbf{B}_\mathcal{P}$

$$p^I(t_1^I, ..., t_k^I) \text{ т. и т. т., когда } p(t_1, ..., t_k) \in I.$$

Характеристическое свойство эрбрановских интерпретаций содержит следующая теорема.

Теорема 1 (Эрбран).
Универсальная теория совместна т. и т.т., когда у нее имеется эрбрановская интерпретация.

Следствие 3. *Конечная теория $\mathcal{P}$ совместна т. и т.т., когда множество простых дизъюнктов (или простых предложений) $\mathcal{P}^s$, получаемых из $\mathcal{P}$ сколемизацией, имеет эрбрановскую интерпретацию.*

Не следует думать, что эта теорема указывает алгоритм поиска решений. Из известной теоремы Черча о неразрешимости логики первого порядка вытекает

Следствие 4. *Проблема существования эрбрановской интерпретации для конечных множеств простых дизъюнктов (предложений) алгоритмически неразрешима.*

Однако теорема Эрбрана принципиально упрощает интерпретацию вычислительных задач.

Определение 3. *Пусть* $\mathbf{Trm}$ *– множество всех термов над множеством функторов* $\mathbf{F}$ *и множеством переменных* $\mathbf{Var}$, *и* $\sigma : \mathbf{Var} \to \mathbf{Trm}$ *– некоторое означивание переменных (всюду определенная функция). Назовем* область изменения *σ множество $Dom(\sigma) = \{X \in \mathbf{Var} | \sigma(X) \neq X\}$. σ называется* подстановкой, *если $Dom(\sigma)$ – конечное множество. Подстановка называется* базисной, *если ее значения на переменных, принадлежащих $Dom(\sigma)$, являются базисными термами. Для выражения (т.е. терма или формулы) e в данной сигнатуре через $e \circ \sigma$ обозначается его* частный случай при подстановке *σ, т.е. результат параллельной замены в e для каждой переменной X всех ее свободных вхождений на терм $\sigma(X)$: $e \circ \sigma = e[X_1/\sigma(X_1), ..., X_l/\sigma(X_l)]$. Частный случай является* базисным, *если σ – базисная подстановка. Назовем* базисным решением *вычислительной задачи с условием $\mathcal{P}$ и вопросом ϕ базисную подстановку σ такую, что*

$$\mathcal{P} \Longrightarrow \phi \circ \sigma.$$

Вычислительная задача разрешима, *если она имеет базисное решение.*

Следствие 5. *Пусть $\mathcal{P}$ – конечное множество простых дизъюнктов (предложений) и $\phi = a_1 \& ... \& a_k$, где a_i - атомы, содержащие переменные $\bar{X}$. Тогда следующие утверждения эквивалентны:*
(i) $\mathcal{P} \Longrightarrow \exists \bar{X} \bullet \phi$;
(ii) вычислительная задача $\mathcal{P} \Longrightarrow \exists \bar{X} \bullet \phi$ разрешима;
(iii) не существует эрбрановской модели системы формул $\mathcal{P} \cup \{\forall \bar{X} \bullet \neg\phi\}$.

Можно гарантировать лишь *частичную разрешимость* вычислительных задач в следующем смысле:

Теорема 2. *Существует частичный алгоритм, который завершается, будучи применен к вычислительной задаче $\mathcal{P} \Longrightarrow \exists \bar{X} \bullet \phi$, тогда и только тогда, когда она разрешима, при этом алгоритм строит некоторое решение.*

Ниже мы опишем частичное решение в этом духе вычислительных задач из очень широкого класса. В целом же приведенные утверждения показывают, что для формулировки и решения вычислительных задач можно без потери общности ограничиться конечными системами простых дизъюнктов или простых предложений над эрбрановскими интерпретациями. Мы выделим это подмножество языка первого порядка, незначительно изменив стандартный синтаксис.

1.3 Хорновские дизъюнкты и логические программы [3]

I.*Синтаксис языка логических программ.*

Пусть выбрана сигнатура первого порядка **P**, **F** и множество предметных переменных **Var**. Тем самым, зафиксированы множество термов **Trm**, эрбрановский универсум **H**, эрбрановский базис **B** и множество всех атомов **A**.

Предложение. Предложение имеет вид

$$h_1; ...; h_k \ :- \ b_1, ..., b_l.$$

где $h_i, b_j \in \mathbf{A}$. $h_1; ...; h_k$ – *голова* предложения, $b_1, ..., b_l$ – его *тело*.
При $k = 0$ предложение называется *целью*. Частный случай цели при $l = 0$ обозначается $\Box$ и называется *противоречием*.
При $l = 0, k > 0$ предложение называется *фактом* и записывается в виде

$$h_1; ...; h_k.$$

Важный частный случай предложений составляют так называемые *дефинитные* или *хорновские* предложения, в которых $k = 1$.

Логический модуль. Конечное множество $\mathcal{P}$ предложений, не содержащее целей, называется *логическим модулем*. Если все предложения $\mathcal{P}$ дефинитны, то и сам модуль называется *дефинитным*.

Логическая программа. *Логической программой* называется логический модуль $\mathcal{P}$, объединенный с некоторой целью $:- a_1, ..., a_r$. Логическая программа *дефинитна*, если ее логический модуль дефинитен.

Логическая процедура. Если логический модуль дефинитен, то каждое его предложение имеет вид

$$p(t_1, .., t_n) \ :- \ b_1, ..., b_l. \ \ (l \geq 0)$$

и, таким образом, *определяет* некоторый (фиксированный для этого предложения) предикат p/n. Множество всех предложений этого модуля, определяющих один и тот же предикат p/n, называется *[логической] процедурой* p/n.

Пример 3. *Пример дефинитной логической программы.*

$append([\,], L, L).$
$append([E|L1], L2, [E|L]) \ :-$
$\quad append(L1, L2, L).$
$prefix(P, L) \ :-$
$\quad append(P, L', L).$
$:- \ prefix([a, X, Y], [Z, b, c, d]]).$

[3] Материал этого и следующего разделов отчасти соответствует главе 1 книги [3].

II.*Семантика языка логических программ.*

II.1.*Трансляционная семантика.*

Определим следующую трансляцию fo предложений логической программы (модуля) $\mathcal{P}$ во множество простых предложений логики первого порядка:
При $k,l>0$

$$(h_1;...;h_k \ :- \ b_1,...,b_l)^{fo} = \forall \bar{X} \bullet (b_1\&...\&b_l \longrightarrow h_1 \vee ... \vee h_k).$$

При $k>0, l=0$

$$(h_1;...;h_k)^{fo} = \forall \bar{X} \bullet (h_1 \vee ... \vee h_k).$$

При $l>0, k=0$

$$(\ :- \ b_1,...,b_l)^{fo} = \forall \bar{X} \bullet \neg(b_1\&...\&b_l).$$

$$\Box^{fo} = \mathbf{1} \longrightarrow \mathbf{0}.$$

Описанная трансляция позволяет говорить о моделях и эрбрановских моделях логической программы $\mathcal{P}$, определяя их через ее трансляцию $\mathcal{P}^{fo}$. Как видим, в этой семантике связка " :- " соответствует импликации, связка "," – конъюнкции, и связка ";" – дизъюнкции. Однако семантику логических программ можно определить и непосредственно.

II.2.*Собственная семантика.*

Обозначим через $ground(\mathcal{P})$ множество всех базисных частных случаев предложений $\mathcal{P}$. Заметим, что это множество конечно т. и т. т., когда $\mathcal{P}$ не использует функторов, отличных от констант.
Итак, пусть I – некоторая эрбрановская интерпретация $\mathcal{P}$ с эрбрановским универсумом $\mathbf{H}_{\mathcal{P}}$ и эрбрановским базисом $\mathbf{B}_{\mathcal{P}}$. Тогда:

1. Для $A_1,...,A_k \in \mathbf{B}_{\mathcal{P}}, k \geq 1$, $I \models A_1,...,A_k$, если $A_i \in I$, $1 \leq i \leq k$;

2. Для $r = (h_1;...;h_k \ :- \ b_1,...,b_l) \in ground(\mathcal{P})$ $I \models r$, если из $I \models b_1,...,b_l$ следует, что найдется такое $j, 1 \leq j \leq k$, для которого верно $I \models h_j$.
(В частности, если $k=0, l>0$, это означает, что $I \not\models b_i$ для некоторого i. Тем самым, $\Box$ является в этой семантике противоречивым предложением.)

3. Для предложения $r \in \mathcal{P}$ $I \models r$, если для любого его базисного частного случая $r' \in ground(r)$ $I \models r'$.

4. Наконец, $I \models \mathcal{P}$, если $I \models r$ для каждого предложения $r \in \mathcal{P}$. Эрбрановская интерпретация I, обладающая этим свойством, называется *моделью* $\mathcal{P}$.
Эти две семантики эквивалентны для эрбрановских интерпретаций.

Предложение 2. *Для всякой логической программы (модуля) $\mathcal{P}$ и для любой эрбрановской интерпретации I $I \models \mathcal{P}$ т. и т. т., когда $I \models \mathcal{P}^{fo}$ в логике первого порядка.*

Это позволяет перенести в собственную семантику понятие решения.

Определение 4. *Базисная подстановка σ является* базисным решением *логической программы*

$$\mathcal{P} \cup \{ \ :- \ A_1,...,A_r\},$$

если каждая модель $M \models \mathcal{P}$ *содержит все атомы* $A_i \circ \sigma, 1 \leq i \leq r$. *Эта программа* разрешима, *если она имеет базисное решение.*

Понятно, что имеет место

Предложение 3. *Пусть* $\mathcal{P} \cup \{ \text{ :- } A_1, ..., A_r\}$ *– логическая программа. Базисная подстановка* σ *является ее базисным решением т. и т. т., когда* $\mathcal{P}^{fo} \models (A_1 \& ... \& A_r) \circ \sigma$ *в логике первого порядка.*

Логические программы имеют некоторые семантические особенности. Так, всякий логический модуль совместен. Его моделью является хотя бы его эрбрановский базис, что неудивительно, если принять во внимание, что логические модули позитивны. Поэтому интерес представляют минимальные модели логических модулей, т.е. модели, собственные подмножества которых уже не являются моделью модуля. Содержательно понятно, что минимальная модель отражает предполагаемый "смысл" модуля. Например, модуль *append* из примера 3 имеет минимальную модель $M_{append} = \{append(L_1, L_2, L) \mid L - \text{конкатенация } L_1, L_2\}$. Однако минимальная модель не всегда единственна, как показывает следующий пример.

Пример 4. $\mathcal{P}_0 = \{p(a); q(a)\}$. *Этот модуль имеет две минимальные модели* $\{p(a)\}$ *и* $\{q(a)\}$. *Хуже того, ни одна из логических программ* $\mathcal{P}_0 \cup \{ \text{ :- } p(a)\}, \mathcal{P}_0 \cup \{ \text{ :- } q(a)\}$ *не разрешима, т.к. согласно определению всякое базисное решение* $A \circ \sigma$ *логической программы* $\mathcal{P}_0 \cup \{ \text{ :- } A\}$ *должно принадлежать пересечению всех моделей* $\mathcal{P}_0$, *которое в данном случае пусто.*

Чтобы избежать эти семантические трудности, мы, впервые поступаясь общностью, ограничимся исключительно дефинитными логическими модулями и программами. В качестве компенсации за потерю общности мы получим класс программ с очень ясной и элегантной семантикой и очень эффективной интерпретацией. Важно то, что именно этот класс приведет нас к определению языка логического программирования Пролог.

1.4 Элементы теории моделей дефинитных логических программ

Начиная с этого момента, мы будем рассматривать исключительно дефинитные логические программы, и для краткости будем опускать определение "дефинитные".

1.4.1 Наименьшая модель и решения задач

Пусть $\mathcal{P}$ – логичекий модуль. Положим

$$M_{\mathcal{P}} = \{A \in \mathbf{B} \mid \mathcal{P}^{fo} \Longrightarrow A\}.$$

Согласно предложению 3 $M_{\mathcal{P}} = \{A \circ \sigma \mid A - \text{атом и } \sigma - \text{базисное решение } \mathcal{P} \cup \{ \text{ :- } A\}\}$. Оказывается, $M_{\mathcal{P}}$ является наименьшей из всех моделей $\mathcal{P}$.

Теорема 3. *(О наименьшей модели)*
(1) $M_{\mathcal{P}} \models \mathcal{P}$.
(2) Пусть $M \subseteq \mathbf{B}$ *и* $M \models \mathcal{P}$. *Тогда* $M_{\mathcal{P}} \subseteq M$.

Поскольку M – наименьшая модель, она единственна. Эта модель вполне может оказаться и пустой.

Пример 5. $\mathcal{P}_0 = \{p \ :\!-\ q\}$. $M_{\mathcal{P}_0} = \{A \in \mathbf{B}_{\mathcal{P}_0} \mid \mathcal{P}_0^{fo} \models A\} = \emptyset$. $\emptyset \models \mathcal{P}_0$.

Наименьшая модель является общей частью всех моделей.

Предложение 4. *Для всякого логического модуля* $\mathcal{P}$ $M_{\mathcal{P}} = \bigcap\{M \subseteq \mathbf{B}_{\mathcal{P}} \mid M \models \mathcal{P}\}$.

Пример 6. *Рассмотрим логический модуль* $Member$

```
   member(E,[E|T]).
% элемент E принадлежит всякому списку, начинающемуся с E
   member(E,[E1|L]) :-
     member(E,L).
% если E принадлежит списку L, то он принадлежит и всякому списку,
% получающемуся из L добавлением нового первого элемента
```

В нем нет констант, но мы предполагаем наличие констант $c1, ..., ck$, $k > 0$. *Таким образом,* $\mathbf{B}_{Member} = \{member(E1, E2) \mid E1, E2 \in \mathbf{lst}(\{c1, ..., ck\}, \emptyset)\}$. *При этом* $M_{Member} = \{member(E1, E2) \mid E1, E2 \in \mathbf{lst}(\{c1, ..., ck\}, \emptyset),\ E1 \in E2\}$.

Наименьшая модель имеет еще одно замечательное эквивалентное определение.

Определение 5. *Пусть* $\mathcal{P}$ *– логичекий модуль и* $A \in \mathbf{B}_{\mathcal{P}}$. *Конечное дерево* $t_{\mathcal{P}}(A)$ *назовем* деревом вывода *атома* A, *если вершинами* $t_{\mathcal{P}}(A)$ *являются атомы из* $\mathbf{B}_{\mathcal{P}}$, *корнем* $t_{\mathcal{P}}(A)$ *является* A, *и для всякой его вершины* H *существует предложение*

$$(H \ :\!-\ B_1, ..., B_l) \in ground(\mathcal{P}),\ l \geq 0,$$

такое, что $\{B_1, ..., B_l\}$ *совпадает со множеством сыновей* H.

Лемма 2. $M_{\mathcal{P}} = \{A \in \mathbf{B}_{\mathcal{P}} \mid$ существует дерево вывода $t_{\mathcal{P}}(A)\}$.

1.4.2 Структура моделей логического модуля

Поскольку каждое подмножество эрбрановского универсума логического модуля $\mathcal{P}$ является его эрбрановской интерпретацией, множество всех эрбрановских интерпретаций, частично-упорядоченное отношением теоретико-множественного включения, образует полную решетку $\mathcal{L}_{\mathcal{P}}$. В этой решетке для любого семейства интерпретаций $M_i \subseteq \mathbf{B}_{\mathcal{P}}$, $i \in J$, $\bigcap_{i\in J} M_i = inf\{M_i \mid i \in J\}$ и $\bigcup_{i\in J} M_i = sup\{M_i \mid i \in J\}$. Как было замечено, множество эрбрановских моделей $\mathcal{P}$ непусто. Поскольку из определения собственной семантики непосредственно следует, что пересечение любого множества эрбрановских моделей $\mathcal{P}$ также является эрбрановской моделью $\mathcal{P}$, подрешетка $\mathcal{M}_{\mathcal{P}}$ эрбрановских моделей $\mathcal{P}$ является нижней подполурешеткой $\mathcal{L}_{\mathcal{P}}$, и наименьшая модель $M_{\mathcal{P}}$ является в ней нижним элементом. Объединение эрбрановских моделей не обязательно является эрбрановской моделью, как показывает следующий пример.

Пример 7. *Рассмотрим логический модуль* $\{a \ :\!-\ d, e.\ b \ :\!-\ c.\}$. *Нетрудно видеть, что* $M_1 = \{d\} \models \mathcal{P}$ *и* $M_2 = \{e\} \models \mathcal{P}$, *однако* $M_1 \cup M_2 = \{d, e\} \not\models \mathcal{P}$. *Между тем,* $sup\{M_1, M_2\} = \{a, d, e\} \models \mathcal{P}$.

Однако нетрудно показать, что $\mathcal{M}_{\mathcal{P}}$ является полной решеткой, и при этом $sup\{M_i \mid M_i \in \mathcal{M}_{\mathcal{P}}, i \in J\} = \bigcap\{F \in \mathcal{M}_{\mathcal{P}} \mid (\bigcup_{i\in J} M_i) \subseteq F\}$. Замечательным свойством наименьшей модели является то, что она может быть "вычислена" как предел последовательности степеней очень естественного монотонного оператора на решетке интерпретаций $\mathcal{L}_{\mathcal{P}}$, непосредственно связываемого с модулем $\mathcal{P}$.

1.4.3 Оператор непосредственного следования и наименьшая модель

Определение 6. *Пусть* $I \subseteq \mathbf{B}_{\mathcal{P}}$. *Положим*

$$T_{\mathcal{P}}(I) = \{h \mid \exists (h \ :- \ b_1, ..., b_k) \in ground(\mathcal{P}) \bullet b_1, ..., b_k \in I\}.$$

Определяемый таким образом оператор $T_{\mathcal{P}}$ *на* $\mathcal{L}_{\mathcal{P}}$ *называется* оператором непосредственного следования.

Очевидно, что оператор непосредственного следования монотонен, т.е. для любых эрбрановских интерпретаций $I_1 \subseteq I_2$ верно $T_{\mathcal{P}}(I_1) \subseteq T_{\mathcal{P}}(I_2)$. Классическая теорема Тарского - Кнастера утверждает, что каждый такой оператор T имеет *наименьшую неподвижную точку*, т.е. наименьший элемент решетки $lfp(T)$, являющийся решением уравнения $lfp(T) = T(lfp(T))$.

Теорема 4. *(О наименьшей неподвижной точке)*
Если $\mathcal{L}$ *– полная решетка и* $T : \mathcal{L} \to \mathcal{L}$ *– монотонный оператор, то существует его наименьшая неподвижная точка* $lfp(T) = inf\ S^{\leq} = inf\ S^{=}$, *где* $S^{\leq} = \{x \in \mathcal{L} \mid T(x) \leq x\}$ *и* $S^{=} = \{x \in \mathcal{L} \mid T(x) = x\}$.

Свяжем с нижним элементом $\perp$ полной решетки $\mathcal{L}$ следующую цепь *степеней* оператора T:

$$\begin{cases} T \uparrow 0 & = \perp \\ T \uparrow (i+1) & = T(T \uparrow i) \\ T \uparrow \omega & = sup\{T \uparrow i \mid i \geq 0\}. \end{cases}$$

Предел этой цепи $T \uparrow \omega$ всегда удовлетворяет неравенству $T \uparrow \omega \leq lfp(T)$. Обратное неравенство выполняется не для каждого монотонного оператора на полной решетке. Однако ему удовлетворяет любой *непрерывный* оператор на полной решетке, т.е. оператор, удовлетворяющий свойству: для всякой цепи $U = \{x_1, x_2, ...| \forall i \bullet x_i \leq x_{i+1}\} \subseteq \mathcal{L}$

$$T(sup\ U) = sup\ \{T(x) \mid x \in U\}.$$

Конечно же, каждый непрерывный оператор на полной решетке монотонен. Поэтому из теоремы 4 следует, что каждый непрерывный ореатор T на полной решетке имеет наименьшую неподвижную точку, вычисляемую как предел цепи его степеней: $lfp(T) = T \uparrow \omega$. Замечательные свойства оператора непосредственного следования вытекают из того, что он непрерывен.

Предложение 5. *Для всякого логического модуля* $\mathcal{P}$ *оператор непосредственного следования* $T_{\mathcal{P}}$ *непрерывен на полной решетке его эрбрановских интерпретаций* $\mathcal{L}_{\mathcal{P}}$.

А теперь заметим, что из определений оператора непосредственного следования и собственной семантики непосредственно вытекает

Предложение 6. *Эрбрановская интерпретация* M *тогда и только тогда является эрбрановской моделью* $\mathcal{P}$, *когда* $T_{\mathcal{P}}(M) \subseteq M$.

Поэтому имеет место следующая теорема, проливающая свет на семантику логических модулей:

Теорема 5. *(О семантике логических модулей)*
Для всякого логического модуля $\mathcal{P}$ $M_{\mathcal{P}} = lfp(T_{\mathcal{P}}) = T_{\mathcal{P}} \uparrow \omega$.

Приведем простой пример, иллюстрирующий вычисление наименьшей модели логического модуля как предела цепи степеней его оператора непосредственного следования.

Пример 8. *Простейшим является случай, когда логический модуль состоит из одних фактов. Например, простейший модуль* $Parent$, *описывающий часть отношений "ребенок – родитель" в некоторой семье,*

$$Parent = \left\{ \begin{array}{l} parent(\text{вася}, \text{коля}). \\ parent(\text{вася}, \text{маша}). \\ parent(\text{вера}, \text{маша}). \\ parent(\text{таня}, \text{коля}). \\ parent(\text{таня}, \text{сережа}). \end{array} \right.$$

совпадает со своей наименьшей моделью M_{Parent}. *При этом* $T_{Parent} \uparrow 1 = Parent$. *В самом деле, в нашей полной решетке нижним элементом является пустое множество. Поэтому для любого логического модуля* $\mathcal{P}$ $T_{\mathcal{P}} \uparrow 1$ *совпадает со множеством фактов в* $ground(\mathcal{P})$. *Добавим к модулю* $Parent$ *определение отношения* $sibling$, *означающего "быть братом или сестрой":* $Sibling = Parent \cup \{r_1\}$, *где* r_1 *– предложение*
$r_1 :$ $sibling(X, Y)$:–
$parent(Z, X)$,
$parent(Z, Y)$,
$X \backslash= Y$.
и будем считать, что отношение несовпадения $\backslash=$ *является встроенным, т.е. в нашем случае попросту описывается тридцатью фактами:* $M^{\backslash=} = \{\text{вася} \backslash= \text{маша}, ...\}$.

Теперь, очевидно,
$T_{Sibling} \uparrow 0 = \emptyset$,
$T_{Sibling} \uparrow 1 = Parent \cup M^{\backslash=}$, *и*
$T_{Sibling} \uparrow 2 = Parent \cup M^{\backslash=} \cup M_{sb}$, *где* $M_{sb} =$
$\{sibling(\text{коля}, \text{маша}), sibling(\text{маша}, \text{коля}), sibling(\text{коля}, \text{сережа}), sibling(\text{сережа}, \text{коля})\}$.

Наконец, опишем модуль $Brother$, *определив его как:* $Brother = Sibling \cup \{r_2, r_3, r_4, r_5\}$, *где*
$r_2 :$ $brother(X, Y)$:–
$sibling(X, Y)$,
$male(X)$.
$r_3 :$ $male(\text{коля})$.
$r_4 :$ $male(\text{вася})$.
$r_5 :$ $male(\text{сережа})$.

Очевидно, что
$T_{Brother} \uparrow 0 = \emptyset$,
$T_{Brother} \uparrow 1 = Parent \cup M^{\backslash=} \cup \{r_3, r_4, r_5\}$,

$T_{Brother} \uparrow 2 = T_{Sibling} \uparrow 2 \cup \{r_3, r_4, r_5\}$,
$T_{Brother} \uparrow 3 = T_{Brother} \uparrow 2 \cup \{brother(\text{коля}, \text{маша}), brother(\text{коля}, \text{сережа}), brother(\text{сережа}, \text{коля})\}$,
u
$T_{Brother} \uparrow 4 = T_{Brother} \uparrow 3 = T_{Brother} \uparrow \omega = M_{Brother}$.

2 SLD-резолюция: операционная семантика логических программ [4]

Идея, лежащая в основе правила интерпретации дефинитных логических программ состоит в том, что подстановка-решение вычислительной задачи вычисляется как композиция локальных подстановок, однозначно связываемых с отдельными предложениями. Эти локальные подстановки-унификаторы в практических ситуациях оказываются просто вычислимыми.

2.1 Синтаксическая унификация

Нас будут интересовать системы уравнений вида

$$\Sigma = \{e_{1i} = e_{2i} \mid 1 \le i \le n\}$$

в которых e_{ij} являются термами (атомами). *Решением* или *унификацией* такой системы является подстановка σ, обращающая в равенство каждое уравнение: $e_{1i} \circ \sigma = e_{2i} \circ \sigma$, $1 \le i \le n$. Конечно, не каждая система имеет решение. Простейший пример: $f(X, a) = g(c, Y)$. Несколько менее очевидный пример: $f(X) = f(h(X))$. Между тем, ясно, что для любой системы Σ множество ее решений $S(\Sigma)$ либо пусто, либо счетно. Однако в случае, когда множество решений бесконечно, в нем можно выделить одно из них, в некотором смысле наиболее общее. Для этого на множестве подстановок вводится следующий естественный порядок.
Определим *композицию* подстановок σ, λ соотношением $(\sigma \circ \lambda)(X) = \sigma(X) \circ \lambda$ для всех $X \in Var$. Эта операция определяет *моноид* подстановок с единицей $\mathbf{1}_\circ$, которой является тождественная подстановка ε. Положим $\sigma_1 \preceq \sigma_2$, если имеется подстановка λ такая, что $\sigma_2 = \sigma_1 \circ \lambda$. Нетрудно показать, что отношение $\preceq$ рефлексивно и транзитивно. Однако оно не антисимметрично (например, $\sigma_1(X) = Y$, $\sigma_2(Y) = X$). Между тем, $\sigma_1 \preceq \sigma_2$, $\sigma_2 \preceq \sigma_1$ означает, что σ_1 и σ_2 совпадают с точностью до переименования переменных. Точнее, назовем подстановку σ *переименованием*, если она является биекцией на Var. Понятно, что переименования образуют группу относительно композиции. Поэтому отношение $\sigma_1 \sim \sigma_2 \Leftrightarrow \exists \lambda \bullet (\sigma_2 = \sigma_1 \circ \lambda \ \& \ (\lambda - \text{переименование}))$ является эквивалентностью. Имеет место следующее утверждение.

Предложение 7. $\sigma_1 \preceq \sigma_2 \ \& \ \sigma_2 \preceq \sigma_1 \Leftrightarrow \sigma_1 \sim \sigma_2$.

Например, для подстановок $\theta(X) = f(Y)$, $\theta(Y) = c$ и $\sigma(X) = f(b)$, $\sigma(Y) = c$, $\sigma(Z) = a$ верно $\theta \preceq \sigma$, так как $\sigma = \theta \circ \lambda$ для $\lambda(Y) = b$, $\lambda(Z) = a$.

Для выражения (терма, атома, формулы, предложения) e его частный случай $e_1 = e \circ \sigma$

[4] Материал этого раздела примерно соответствует главе 2 книги [3].

при переименовании σ называется *вариантом* e. Понятно, что при этом $e = e_1 \circ \sigma^{-1}$, то есть и e является вариантом e_1. Таким образом, отношение "быть вариантом" на множестве выражений является эквивалентностью, и мы сохраним за ним обозначение $\sim$.

Определение 7. *Решение σ системы уравнений Σ является ее* наиболее общим унификатором (НОУ), *если для любого другого ее решения σ' верно $\sigma \preceq \sigma'$. Подстановка σ является НОУ системы термов (атомов) $e_1, e_2, ..., e_n$ $(n > 1)$, если σ является НОУ системы уравнений $\{e_1 = e_2, ..., e_1 = e_n\}$.*

Например, система уравнений $f(X, g(X, X)) = f(Y, Z)$ разрешима, и подстановки $\sigma_1(X) = Y, \sigma_1(Y) = a, \sigma_1(Z) = g(Y, Y)$ и $\sigma_2(X) = Y, \sigma_2(Z) = g(Y, Y)$ являются ее решениями, но σ_2 является НОУ, тогда как σ_1 НОУ не является.

Определение 8. *Система уравнений Σ является* разрешенной, *если она имеет вид $\{X_1 = e_1, ..., X_n = e_n\}$, переменные $X_1, ..., X_n$ попарно различны, и ни одна из них не имеет вхождений в термы $e_1, ..., e_n$. Обозначим через $\overset{\mapsto}{\Sigma}$ подстановку $\{X_1 \mapsto e_1, ..., X_n \mapsto e_n\}$, определяемую этой системой.*

Опишем простое исчисление Ru для приведения разрешимых систем уравнений к разрешенной форме.

$R1.$ $e = e,\ \Sigma \implies \Sigma,$
$R2.$ $f(t_1, ..., t_k) = f(u_1, ..., u_k),\ \Sigma \implies t_1 = u_1, ..., t_k = u_k, \Sigma,$
$R3.$ $e = X,\ \Sigma \implies X = e,\ \Sigma,$
если $(e \notin Var)$
$R4.$ $X = e,\ \Sigma \implies X = e,\ \Sigma \circ \{X \mapsto e\},$
(если $X \in Var(\Sigma) \setminus Var(e)$).

Вывод в Ru - это последовательность систем уравнений $\Sigma_1 \implies \Sigma_2 \implies ... \implies \Sigma_m$, в которой каждая система получается из предыдущей одним из правил $R1 - R4$ (при наличии такого вывода Σ_m *выводима* из Σ_1, обозначение: $\Sigma_1 \Longrightarrow^* \Sigma_m$). Вывод является *завершенным*, если его нельзя продолжить.

Исчисление Ru обладает замечательными свойствами:

Теорема 6.
1. Правила $R1 - R4$ сохраняют решения, т.е. из $\Sigma \implies \Sigma'$ следует $S(\Sigma) = S(\Sigma')$.
2. Для любых двух завершенных выводов $\Sigma \Longrightarrow^* \Sigma_1$, $\Sigma \Longrightarrow^* \Sigma_2$ $S(\Sigma_1) \neq \emptyset$ т. и т.т., когда системы Σ_1, Σ_2 являются разрешенными и $\overset{\mapsto}{\Sigma_1} \sim \overset{\mapsto}{\Sigma_2}$.
3. Для каждой системы Σ существует такая константа $c > 0$, что длины всех выводов ограничены сверху величиной $c^{|\Sigma|}$ ($|\Sigma|$ обозначает длину записи Σ).

Таким образом, можно недетерминированно (наугад) применять правила исчисления, пока не будет получен завершенный вывод. Если его заключительная система уравнений является разрешенной, то мы получаем некоторый НОУ. В противном случае исходная система неразрешима (или, как говорят, *не унифицируема*).

Недетерминированный алгоритм унификации.

Вход: Система уравнений Σ.
ПОКА для текущей системы Σ_1
СУЩЕСТВУЕТ система Σ_2, **ТАКАЯ ЧТО**
$\Sigma_1 \implies \Sigma_2$,
ВЫПОЛНЯТЬ $\Sigma_1 := \Sigma_2$;
КОНЕЦ_ПОКА
ЕСЛИ система Σ_1 разрешена,
ТО Выход $= \overset{\mapsto}{\Sigma_1}$
ИНАЧЕ неудача
КОНЕЦ_ЕСЛИ

Детерминированный алгоритм унификации получается, если рассматривать упорядоченные системы уравнений, и выбирать в них первое уравнение, к которому применимо некоторое правило. Приведем два примера.

Пример 9. $\Sigma = \{f(X, g(X,X), U, g(U,U)) = f(c, Z, g(Z,Z), V)\}$
$\Longrightarrow_{R2}$
$\{X = c,\ g(X,X) = Z,\ U = g(Z,Z),\ g(U,U) = V\}$
$\overset{*}{\Longrightarrow}_{R4,R3}$
$\{X = c,\ Z = g(c,c),\ U = g(Z,Z),\ g(U,U) = V\}$
$\overset{*}{\Longrightarrow}_{R4,R3}$
$\{X = c,\ Z = g(c,c),\ U = g(g(c,c), g(c,c)),\ g(U,U) = V\}$
$\overset{*}{\Longrightarrow}_{R4,R3}$
$\{X = c,\ Z = g(c,c),\ U = g(g(c,c), g(c,c)),\ V = g(g(g(c,c), g(c,c)), g(g(c,c), g(c,c)))\}$.

Этот пример показывает, что в худшем случае алгоритм унификации затрачивает экспоненциальные время и память.

Пример 10. $\Sigma = \{f(X, g(X,X)) = f(u(a), g(u(b), Z))\}$
$\Longrightarrow_{R2}$
$\{X = u(a),\ g(X,X) = g(u(b), Z)\}$
$\Longrightarrow_{R4}$
$\{X = u(a),\ g(u(a), u(a)) = g(u(b), Z)\}$
$\Longrightarrow_{R2}$
$\{X = u(a),\ u(a) = u(b),\ u(a) = Z\}$
$\Longrightarrow_{R2}$
$\{X = u(a),\ a = b,\ u(a) = Z\}$
Несовместная система. Неудача.

2.2 Интерпретатор логических программ

Теперь, располагая алгоритмом унификации, мы в состоянии описать недетерминированный интерпретатор логических программ, задающий операционную семантику логических программ. Интерпретатор удобно представлять себе как абстрактную машину с бесконечным множеством состояний. Переход из одного состояния в другое определяется так называемым правилом SLD-резолюции.

Определение 9. *Пусть* $\mathcal{P} \cup \{ :- G\}$ *- логическая программа, и* $\mathbf{H} = \mathbf{H}_\mathcal{P}$, $\mathbf{B} = \mathbf{B}_\mathcal{P}$, $\mathbf{Trm}$ *и* $\mathbf{A}$ *- связываемые с нею эрбрановский универсум, эрбрановский базис, множество термов и множество всех атомов.* Состояние *интерпретатора это - пара вида* $< :- Goal;\ \sigma >$, *в которой первый элемент* $:- Goal$ *- некоторая цель, состоящая из атомов из* $\mathbf{A}$, *а второй элемент - некоторая подстановка со значениями из* $\mathbf{Trm}$. Начальным *является состояние* $< :- G;\ \varepsilon >$.

Переходы между состояниями определяются отношением $s_1 \overset{r,\theta}{\vdash} s_2$, *где*

- $s_1 =< :- G_1;\ \sigma_1 >$, $G_1 = A_1, ..., A_i, ..., A_k$, $s_2 =< :- G_2;\ \sigma_2 >$, $G_2 = A_1, ..., \bar{B}, ..., A_k$,
- $r \sim (H :- \bar{B}) \in \mathcal{P}$, *(т.е. r - вариант предложения из $\mathcal{P}$)*
- $A_i \circ \sigma_1 \circ \theta = H \circ \theta$,
- $\sigma_2 = \sigma_1 \circ \theta$.

Отношение $\overset{r,\theta}{\vdash}$ *называется* обобщенной SLD-резолюцией, A_i - устраняемой подцелью, r - устраняющим предложением, θ - *устраняющей подстановкой, цель* $:- G_2$ - резольвентой *цели* $:- G_1$ *и предложения* r. *Состояние вида* $< \Box,\ \theta >$ *называется* заключительным.

Последовательность вида

$$\beta = (s_0 \overset{r_1,\theta_1}{\vdash} s_1 \overset{r_2,\theta_2}{\vdash} , ..., \overset{r_m,\theta_m}{\vdash} s_m, ...)$$

в которой $s_0 =< :- G_0;\ \theta_0 >$ $(G_0 = G,\ \theta_0 = \varepsilon)$ *- начальное состояние, и для всякого* $i > 0$ $s_i =< :- G_i, \sigma_i >$, $\sigma_i = \theta_0 \circ ... \circ \theta_i$, *и*

$$Var(r_i) \cap \cup_{j=0}^{i-1}(Var(C_j) \ \cup \ Var(r_j)) = \emptyset, \qquad (\#)$$

называется обобщенным вычислением $\mathcal{P}$ *с* входной последовательностью $r_1, ..., r_m$. *Если β - конечна и состояние s_m - заключительное, то β называется* обобщенным опровержением *с* вычислимой подстановкой $\theta_1 \circ ... \circ \theta_m \mid_{Var(G)}$. *Если в β для каждого i подстановка θ_i является НОУ для тождества $A \circ \sigma_i = H$, где A – подцель, устраняемая в G_i, и H – голова устраняющего предложения r_{i+1}, то β называется* опровержением *и подстановка* $\theta_1 \circ ... \circ \theta_m \mid_{Var(G)}$ *называется* вычислимым решением.

Функция F, отображающая начальные отрезки обобщенных вычислений $\beta[j] = (s_0 \overset{r_1,\theta_1}{\vdash} s_1 \overset{r_2,\theta_2}{\vdash} , ..., \overset{r_j,\theta_j}{\vdash} s_j)$, $j \geq 0$ *в натуральные числа* $0 < F(\beta[j]) \leq |s_j|$, *называется* стратегией. *Если в (обобщенном) вычислении (опровержении) β на каждом шаге $j+1$ устраняемой является подцель с номером $F(\beta[j])$, то говорят, что β является $F-$(обобщенным) вычислением (опровержением).*

Приведем несколько примеров вычислений. Чтобы избежать коллизии переменных, то есть обеспечить условие (#), мы воспользуемся тем, что все переменные в предложениях

логических программ связаны, и, таким образом, могут быть переименованы. Мы примем простую дисциплину переименования, помечая все переменные в устраняющем предложении на шаге i верхним индексом i.

Пример 11. *Рассмотрим логическую программу, получаемую добавлением цели* $:- member(2,[0,1|T])$ *к логическому модулю* $Member$ *из примера 6. С этой программой связывается следующее вычисление.*

$< :- member(2,[0,1|T]);\ \varepsilon > \overset{2,\theta_1}{\vdash}$
% здесь 2 указывает на то, что на первом шаге устраняющим является второе
% предложение процедуры member/2, приобретающее при нашей дисциплине
% переименования вид: $member(E^1,[E1^1|L^1]) :- member(E^1,L^1)$, и подстановкой
% служит НОУ: $\theta_1 = \{E^1 \mapsto 2,\ E1^1 \mapsto 0,\ L^1 \mapsto [1|T]\}$
$< :- member(E^1,L^1);\ \theta_1 > \overset{2,\theta_2}{\vdash}$
% новая подцель устраняется копией второго предложения процедуры,
% помеченной номером шага 2; новая устраняющая подстановка $\theta_2 =$
$\%\{E^2 \mapsto 2,\ E1^2 \mapsto 1,\ L^2 \mapsto T\}$ определяется как НОУ, являющийся решением
% уравнения $member(2,[1|T]) = member(E^2,[E1^2|L^2])$, в котором
% $member(2,[1|T]) = member(E^1,L^1) \circ \theta_1$
$< :- member(E^2,L^2);\ \theta_1 \circ \theta_2 > \overset{1,\theta_3}{\vdash}$
% теперь подцель $member(E^2,L^2)$ устраняется копией с номером 3 первого
% предложения процедуры: $member(E^3,[E^3|T^3])$, и устраняющая подстановка –
% это НОУ, являющийся решением уравнения $member(2,T) = member(E^3,[E^3|T^3])$,
% то есть подстановка $\theta_3 = \{E^3 \mapsto 2,\ T \mapsto [2|T^3]\}$
$< \Box;\ \theta_1 \circ \theta_2 \circ \theta_3 >$
% Таким образом, это вычисление, и вычислимой является подстановка
% $\theta_1 \circ \theta_2 \circ \theta_3 \mid_{\{T\}} = \{T \mapsto [2|T^3]\}$.

Эта вычислимая подстановка описывает в общем виде решение рассматриваемой вычислительной задачи. Действительно,

$$Member^{fo} \Longrightarrow \forall T^3 \bullet member(2,[0,1|T]) \circ \{T \mapsto [2|T^3]\}.$$

Как мы вскоре увидим, это не случайно.

Отметим, что на третьем шаге можно было бы вместо первого предложения вновь выбрать устраняющим второе предложение процедуры. В этом случае результат был бы другим:

$< :- member(E^2,L^2);\ \theta_1 \circ \theta_2 > \overset{2,\theta'_3}{\vdash}$
% подцель $member(E^2,L^2)$ устраняется копией с номером 3 второго предложения,
% и в этом случае устраняющая подстановка – это НОУ, являющийся решением
% уравнения $member(2,T) = member(E^3,[E1^3|L^3])$, то есть подстановка $\theta'_3 =$
% $\{E^3 \mapsto 2,\ T \mapsto [E1^3|L^3]\}$
$< :- member(E^3,L^3);\ \theta_1 \circ \theta_2 \circ \theta'_3 > \overset{1,\theta_4}{\vdash}$
% теперь уже подцель $member(E^3,L^3)$ устраняется копией с номером 4 первого

% предложения процедуры: $member(E^4, [E^4|T^4])$, *и устраняющая подстановка –*
% это НОУ, являющийся решением уравнения $member(2, L^3) = member(E^4, [E^4|T^4])$,
% то есть подстановка $\theta_4 = \{E^4 \mapsto 2,\ T \mapsto [E1^3, 2|T^4]\}$
$< \Box;\ \theta_1 \circ \theta_2 \circ \theta'_3 \circ \theta_4 >$
Это новое вычисление дает новую вычислимую подстановку $\{T \mapsto [E1^3, 2|T^4]\}$ *и, соответственно, новое общее решение задачи:*

$$Member^{fo} \Longrightarrow \forall E1^3, T^4 \bullet member(2, [0, 1|T]) \circ \{T \mapsto [E1^3, 2|T^4]\}.$$

Понятно, что таким образом можно получить счетное множество разных решений задачи.

Обобщенные вычисления очень естественно описываются как процесс построения деревьев вычисления. Корнем дерева вычисления является символ исходной цели " :– ", и его непосредственными потомками являются подцели этой цели. Все прочие вершины являются атомами из посылок устраняющих предложений: если устраняемая подцель A отвечает вершине v и устраняется она предложением H :– $B_1, ..., B_k$, то v имеет в дереве вычисления непосредственных потомков $v_1, ..., v_k$, которым отвечают соответственно $B_1, ..., B_k$. Вершины, соответствующие устраненным подцелям, помечаются номером устраняющего предложения в содержащей его процедуре, а также устраняющей подстановкой. Понятно, что подцели в листьях дерева вычисления либо устранены предложениями-фактами (и, следовательно, снабжены устраняющими подстановками), либо еще не устранены, и входят в последнюю цель вычисления. Если такая неустраненная подцель будет устраняться на следующем шаге, то соответствующий ей лист помечается символом $\bullet$ (говорят, что эта подцель *находится в фокусе*). Рассмотрим пример.

Пример 12. *Пусть Appr – логическая программа, содержащая процедуру* $append/3$ *из примера 3, следующую процедуру* $appr/4$:
$appr(L, S, I, O)$:–
$append(P, S, L)$,
$append(P, I, O)$.
и цель
:– $appr([a, b], [b], [c], R)$.
С этой программой связывается следующее вычисление:

$< \quad :- appr([a,b],[b],[c,R]);\ \varepsilon > \overset{1,\theta_1}{\vdash}$
$\quad \% \ \theta_1 = \{L^1 \mapsto [a,b], S^1 \mapsto [b], I^1 \mapsto [c]\}$
$< \quad :- append(P^1, S^1, L^1), append(P^1, I^1, R^1);\ \theta_1 > \overset{2,\theta_2}{\vdash}$
$\quad \% \ \theta_2 = \{P^1 \mapsto [a|L1^2], L2^2 \mapsto [b], L^2 \mapsto [b], E^2 \mapsto a\}$
$< \quad :- append(L1^2, L2^2, L^2), append(P^1, I^1, R^1);\ \theta_1 \circ \theta_2 > \overset{1,\theta_3}{\vdash}$
$\quad \% \ \theta_3 = \{L1^2 \mapsto [\,]\}$
$< \quad :- append(P^1, I^1, R^1);\ \theta_1 \circ \theta_2 \circ \theta_3 > \overset{2,\theta_4}{\vdash}$
$\quad \% \ \theta_4 = \{E^4 \mapsto a, L2^4 \mapsto [c], L1^4 \mapsto [\,], R^1 \mapsto [a|L^4]\}$
$< \quad :- append(L1^4, L2^4, L^4);\ \theta_1 \circ \theta_2 \circ \theta_3 \circ \theta_4 > \overset{1,\theta_5}{\vdash}$

$\%\ \theta_5 = \{L^4 \mapsto [c]\}$
$< \Box :\ \theta_1 \circ \theta_2 \circ \theta_3 \circ \theta_4 \circ \theta_5 > .$

Это вычисление имеет дерево, изображенное на Рис. 1a). При этом начальный отрезок вычисления, соответсвующий его первым двум шагам, имеет дерево, изображенное на Рис. 1b).

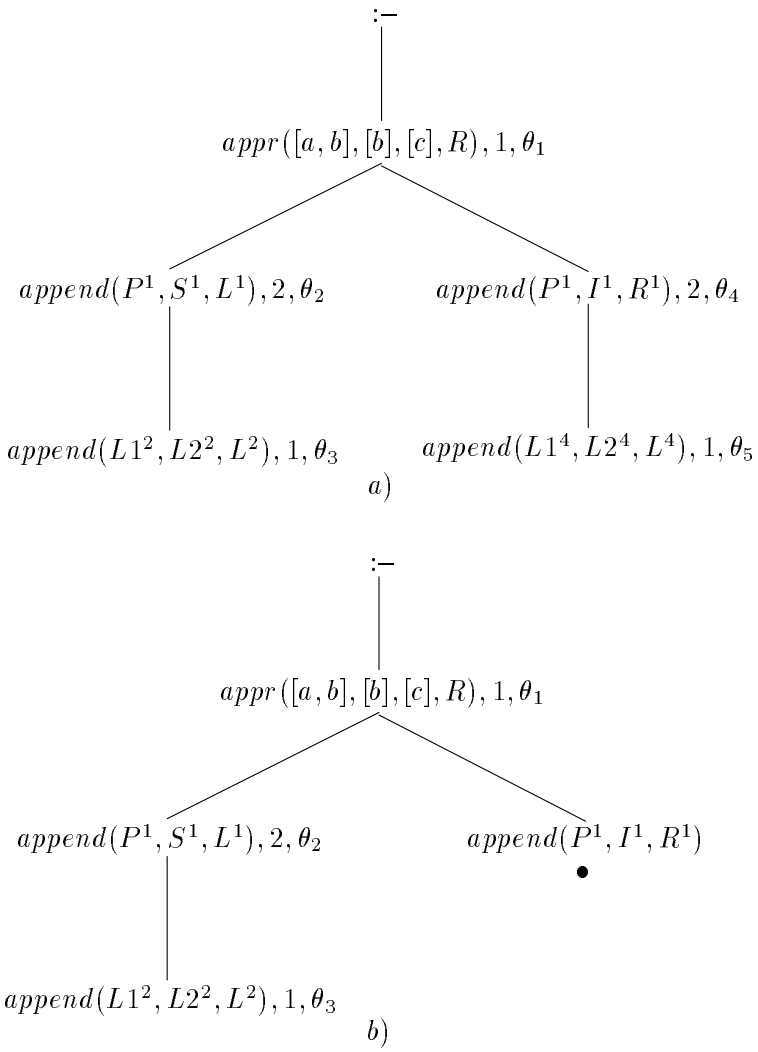


Рис.1 Деревья вычислений.

Итак, будучи применен к логической программе, интерпретатор связывает с ней некоторое множество обобщенных вычислений. Каждое конкретное вычисление может оказаться бесконечным, неудачным (то есть конечным, но непродолжаемым до опровержения), или,

наконец, опровержением. В этом последнем случае, как мы убедились, может статься, что имеется несколько различных опровержений, приводящих к разным решениям.

Пример 13. *Рассмотрим логическую программу*

```
p(X).
p(b) :- p(b).
q(a).
:- p(X),q(X).
```

С этой программой интерпретатор связывает вычисления всех трех типов:
1) бесконечное вычисление:
$(< \ :- p(X), q(X) \ ; \ \varepsilon > \overset{2,\sigma_1}{\vdash} < \ :- p(b), q(b) \ ; \ \sigma_1 > \overset{2,\varepsilon}{\vdash} < \ :- p(b), q(b) \ ; \ \sigma_1 > \overset{2,\varepsilon}{\vdash} \ ...)$,
в котором $\sigma_1(X) = b$;
2) неудачное вычисление:
$(< \ :- p(X), q(X) \ ; \ \varepsilon > \overset{2,\sigma_1}{\vdash} < \ :- p(b), q(b) \ ; \ \varepsilon > \overset{1,\sigma_1}{\vdash} < \ :- q(b) \ ; \ \sigma_1 >)$;
3) опровержение:
$(< \ :- p(X), q(X) \ ; \ \varepsilon > \overset{1,\varepsilon}{\vdash} < \ :- q(X) \ ; \ \varepsilon > \overset{3,\sigma_2}{\vdash} < \Box \ ; \ \sigma_2 >)$,
в котором $\sigma_2(X) = a$.

Такая неопределенность приводит к естественному вопросу о том, какова связь между вычислимыми подстановками и решениями логической программы. Важно то, что интерпретатор логических программ оказывается логически корректным.

Определение 10. *Пусть* $\mathcal{P} \cup \{ \ :- G\}$ *- логическая программа. Подстановка* σ *(не обязательно базисная) является ее* общим решением, *если*

$$\mathcal{P}^{fo} \implies \forall \bar{X} \bullet G^{fo} \circ \sigma$$

(или, что то же самое, $\mathcal{P}^{fo} \models G^{fo} \circ \sigma$*).*

Теорема 7. *(Теорема корректности).*
Для любой логической программы $\mathcal{P} \cup \{ \ :- G\}$ *(и любой стратегии* F*) всякая ее (* F*-) вычислимая подстановка является и ее общим решением.*

В частности, эта теорема верна для (F-) вычислимых решений. Содержательный смысл этой фундаментальной теоремы состоит в том, что интерпретатор логических программ никогда не дает неадекватных результатов. Следует заметить, что этот интерпретатор является недетерминированным, и на каждом шаге вычисления имеет три независимых выбора:

C_1 : выбор вхождения устраняемой подцели (если F не зафиксирована);

C_2 : выбор устраняющего предложения;

C_3 : выбор устраняющей подстановки.

Таким образом, независимо от последовательности сделанных выборов, если вычисление оказывается опровержением, то вычислимая подстановка непременно является общим решением. Неявно отсюда следует, что и любой другой интерпретатор, получающийся из описанного введением некоторого управления выборами $C_1 - C_3$, также будет корректен. Кроме того, эта теорема обнаруживает простую связь между операционной семантикой и семантикой наименьшей модели.

Определение 11. *Обозначим через* $succ(\mathcal{P})$, $succ^*(\mathcal{P})$, $succ^F(\mathcal{P})$, *и* $succ^{F*}(\mathcal{P})$ *множества всех базисных частных случаев* $A \circ \sigma$ *соответственно относительно вычислимых решений, вычислимых подстановок, (* $F-$*) вычислимых решений и (* $F-$*) вычислимых подстановок логических программ вида* $\mathcal{P} \cup \{ :- A\}$, $A \in \mathbf{B}$. *Эти множества называются* (обобщенной) ($F-$) областью успеха *логического модуля* $\mathcal{P}$.

Следствие 6. *Для всякого логического модуля* $\mathcal{P}$ *и для любой стратегии* F $succ^*(\mathcal{P}) \subseteq M_{\mathcal{P}}$, $succ^{F*}(\mathcal{P}) \subseteq M_{\mathcal{P}}$.

Итак, интерпретатор логических программ находит только решения вычислительной задачи, но **все ли решения?** Этот естественный вопрос относится к свойству, обратному свойству корректности, - свойству *полноты* интерпретатора. Оказывается, интерпретатор полон и по отношению к базисным целям, и по отношению к общим целям.

Теорема 8. *(Слабая теорема полноты).*
Для всякого логического модуля $\mathcal{P}$ *и для любой стратегии* F
$succ(\mathcal{P}) = succ^*(\mathcal{P}) = succ^F(\mathcal{P}) = succ^{F*}(\mathcal{P}) = M_{\mathcal{P}}$.

Теорема 9. *(Сильная теорема полноты).*
Для всякой логической программы $\mathcal{P} \cup \{ :- G\}$, *всякого ее общего решения* $\mathcal{P}^{fo} \models G^{fo} \circ \sigma$ *и для любой стратегии* F *всегда найдется не менее общее* $F-$*вычислимое решение* $\lambda \preceq \sigma$.

Обсудим содержательные и формальные следствия из этого фундаментального свойства логических программ.

Во-первых, отметим, что согласно теореме корректности подстановка λ также является общим решением логической программы $\mathcal{P} \cup \{ :- G\}$, то есть $\mathcal{P}^{fo} \models G^{fo} \circ \lambda$. При этом σ является частным случаем λ. Иначе говоря, интерпретатор вычисляет наиболее общие решения. Отсюда, в частности, следует, что все вычислимые решения являются вариантами друг-друга, то есть совпадают с точностью до переименований переменных.

Следствие 7. *Для любых вычислимых решений* λ_1, λ_2 *разрешимой логической программы* $\mathcal{P} \cup \{ :- G\}$ *верно* $\lambda_1 \sim \lambda_2$.

Во-вторых, не должно остаться незамеченным, что в теореме 9 речь идет о вычислимых решениях, а не просто о вычислимых подстановках. Иначе говоря, эта теорема указывает на избыточность недетерминированного выбора устраняющей подстановки C_3 : вместо этого можно всегда обходиться НОУ, вычисляя его детерминированным алгоритмом унификации.

Наконец, в-третьих, согласно теореме 9, и недетерминированный выбор вхождения устраняемой подцели C_3 является избыточным: вместо этого можно раз и навсегда выбрать некоторую одну стратегию F. Теорема гарантирует, что множество решений не зависит от выбора стратегии. Например, можно выбрать стратегию F^l, требующую устранения на каждом шаге первой (самой левой) подцели, или F^r, требующую устранения последней (самой правой) подцели. Таким образом, в условиях теоремы 9 интерпретатор логических программ на каждом шаге вычисления имеет недетерминированный выбор только типа C_2 : конечный выбор применимого устраняющего предложения, то есть предложения, голова которого унифицируема с устраняемой подцелью. Это обстоятельство позволяет очень просто описать все пространство вычислений логической программы $\mathcal{P} \cup \{ :- G\}$ как (вообще говоря, бесконечное) дерево - так называемое *SLD-дерево*. Его вершинами являются

промежуточные состояния вычислений (начальное состояние - общее для всех вычислений - корень дерева). Дуги соответствуют шагам SLD-резолюции, переводящим состояние в начале дуги в состояние в ее конце. Дуг имеется ровно столько, сколько есть различных предложений, применимых к устраняемой подцели. Таким образом, ветви SLD-дерева соответствуют $F-$вычислениям. Ветвь из корня в данную вершину-состояние $v =< \ :- G; \sigma >$ описывает начальный отрезок $F-$вычисления, стратегия F однозначно определяет вхождение устраняемой подцели, и из вершины v выходят дуги в вершины, продолжающие ветвь, если в состоянии v имеются предложения, применимые к устраняемой подцели. Если применимых предложений нет, то вершина либо является заключительным состоянием, и ведущая в нее ветвь соответствует некоторому $F-$вычислению (является успешной), либо является тупиковой, и ведущая в нее ветвь оказывается непродолжаемой (неудачной).

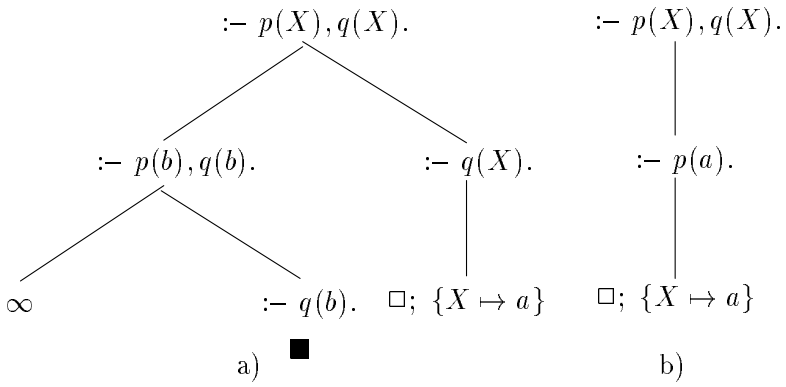


Рис.2 $SLD-$ деревья для различных стратегий.

Любопытно, что при различных стратегиях SLD-деревья могут оказаться разными, как это видно на примере логической программы из примера 13. На рисунке 2a) изображено ее SLD-дерево для стратегии F^l, а на рисунке 2b) - SLD-дерево для стратегии F^r. Мы видим, что в первом SLD-дереве имеется бесконечная ветвь (помеченная ∞), успешная ветвь, и счетное множество неудачных ветвей (помеченных $\blacksquare$), тогда как во втором SLD-дереве имеется единственная ветвь, и она является успешной. При этом оба SLD-дерева описывают одно и то же решение: $\{X \mapsto a\}$.

Теорему 9 можно переформулировать в терминах SLD-деревьев.

Следствие 8. *Логическая программа* $\mathcal{P} \cup \{ \ :- G\}$, *имеет общее решение* $\mathcal{P}^{fo} \models G^{fo} \circ \sigma$ *тогда и только тогда, когда для любой стратегии* F *в SLD-дереве, соответствующем этой стратегии, найдется успешная ветвь. При этом каждая успешная ветвь описывает некоторое* $F-$*вычислимое решение* $\lambda \preceq \sigma$.

Подведем итог. Операционная семантика логических программ, определяемая с помощью интерпретатора на основе правила SLD-резолюции, оказывается эквивалентной их логической семантике. При этом даже интерпретатор, оптимизированный с помощью теоремы 9, остается недетерминированным. Представим себе некий алгоритм, обходящий в определенном порядке SLD-дерево, и сохраняющий свойство полноты, т.е. находящий успешную

ветвь, если таковая имеется. Всякий такой алгоритм может использоваться для автоматического доказательства теорем. Между тем, как нетрудно понять, подобный алгоритм вынужден компенсировать недетерминированность типа C_2, и для этого перебирать (в худшем случае) все применимые устраняющие предложения, иначе говоря, все дуги, исходящие из каждой посещаемой вершины. Такой алгоритм оказывается слишком сложным, чтобы можно было, меняя подходящим образом порядок предложений в процедурах логического модуля, управлять надлежащим образом порядком обхода ветвей (то есть поиском вывода). Именно по этой причине язык логических программ нельзя признать языком программирования. Это язык для формулировки решаемых автоматически вычислительных задач. Ситуация была бы совершенно иной, если бы интерпретатор пользовался неким раз навсегда фиксированным порядком перебора предложений (например, он перебирал бы их в том самом порядке, в котором они присутствуют в логических процедурах). В этом случае интерпретатор стал бы детерминированным. Он соответствовал бы простейшему алгоритму обхода SLD-дерева сверху-вниз, слева-направо. Соответственно, подбирая тот или иной порядок предложений в процедурах, их автор добивался бы определенного порядка перебора применимых предложений, то есть на самом деле управлял бы процессом вычисления решения. Здесь действительно можно было бы говорить о программировании. Казалось бы, в чем проблема, почему бы и не выбрать столь простой интерпретатор? Дело в том, что этот интерпретатор не полон, что видно из следующего простейшего примера.

Пример 14. *Программа loop*

```
r :-
  r.
r.
:- r.
```

С одной стороны, эта логическая программа имеет общее решение ε: $loop^{fo} \models r$, с другой - детерминированный интерпретатор, впадая в цикл, следует вдоль бесконечной левой ветви SLD-дерева, и потому никогда не находит решения, соответствующего любой конечной успешной ветви. Заметим, что, переставив предложения,

```
r.
r :-
  r.
:- r.
```

мы добиваемся, что интерпретатор находит решение на первом же шаге.

Таким образом, добиваясь детерминированности интерпретатора, мы должны сделать сознательный выбор между свойством полноты и автоматическим доказательством теорем с одной стороны, и неполнотой, но возможностью программировать процесс решения задачи - с другой. Мы делаем выбор в пользу последней альтернативы, и приходим к определению языка Пролог.

3 От логических программ к Прологу

3.1 Правило интерпретации Пролога

Язык Пролог имеет только два синтаксических отличия от языка логических программ:

- в каждой процедуре программы зафиксирован линейный порядок предложений [5] (т.е. можно говорить о первом, втором, и т.д. предложении),
- в Прологе могут использоваться так называемые "встроенные" предикаты, т.е. предикаты, не имеющие определений в программах, но включенные в его библиотеку.

В приводимых ниже примерах их семантика объясняется в комментариях.
Вычисление программы на Прологе – это последовательность состояний вида $< \ :\!- G_i;\ \sigma_i >$, в которой каждое следующее состояние получается из предыдущего либо шагом SLD-резолюции, либо шагом возврата. При этом устраняется всегда первая подцель (т.е. используется стратегия F^l), а в качестве стратегии выбора устраняющего предложения используется следующее правило:

Правило выбора устраняющего предложения:
Для устранения подцели A следует выбрать ПЕРВОЕ предложение вида $H \ :\!- B$, в котором H унифицируемо с A в текущем состоянии σ_i и которое ранее не было использовано в вычислении для устранения этой подцели.

Это правило позволяет различать состояния, являющиеся *точками выбора*, и *детерминированные* состояния вычисления. Состояние $< \ :\!- G_i;\ \sigma_i >$ является точкой выбора на шаге

$$< \ :\!- G_i;\ \sigma_i > \overset{\Theta_i, r_i}{\vdash} < \ :\!- G_{i+1};\ \sigma_{i+1} >$$

если предложение r_i не является ПОСЛЕДНИМ в своей процедуре. В противном случае состояние является детерминированным.

Правило интерпретации программы $\mathcal{P} \cup \{ \ :\!- G\}$:
Начальное состояние: $s_1 =< \ :\!- G;\ \varepsilon >$.

1. **Шаг SLD-резолюции:**
В состоянии $s_i =< \ :\!- G_i, \sigma_i >$
если $:\!- G_i = [\,]$, **то Конец.** Результат: $(\sigma_i \mid_{Var(G)})$;
если $G_i = p(\bar{t}), \bar{G}$ **и**
правило выбора устраняющего предложения выбирает $r_i = (p(\bar{u}) \ :\!- \bar{B})$, **то**
следующим становится состояние $s_{i+1} =< \ :\!- G_{i+1};\ \sigma_{i+1} >$, возникающее на шаге SLD-резолюции

$$< \ :\!- G_i;\ \sigma_i > \overset{\Theta_i, \tilde{r}_i}{\vdash} < \ :\!- G_{i+1};\ \sigma_{i+1} >$$

устраняющем подцель $p(\bar{t})$ с помощью НОУ Θ_i и нового варианта предложения $\tilde{r}_i$;
затем выполняется следующий шаг резолюции;

[5] Порядок процедур не является существенным и в реальных программах определяется лишь соображениями удобства.

если же правило выбора НЕ МОЖЕТ выбрать устраняющего предложения [6] , **то**

в состоянии s_i возникает **неудача** и выполняется шаг возврата.

2. **Шаг возврата:**
В состоянии $s_i =< \ :- G_i;\ \sigma_i >$
если $i = 1$, **то Конец. (Неудача вычисления)**
если $i > 1$ **и** s_i – детерминированное состояние, **то**

восстанавливается предыдущее состояние s_{i-1}
(заметим, что при этом отменяется устраняющая подстановка);
выполняется шаг возврата;

если же $i > 1$ и s_i – точка выбора, **то**

устраняющее правило ставится использованным;
отменяется устраняющая подстановка;
выполняется следующий шаг резолюции в состоянии s_i.

□

Как мы видим, интерпретатор Пролога реализует упомянутый выше алгоритм обхода SLD-дерева сверху-вниз, слева-направо, уточненный правилом возврата: встретив тупик, он возвращается в ближайшую вершину, где был сделан выбор устраняющего предложения, и выбирает следующее предложение, если таковое еще имеется (если его нет, то он продолжает возврат, как если бы попал в тупик). Таким образом, успешно завершая вычисление для программы $\mathcal{P} \cup \{ \ :- G\}$, интерпретатор Пролога находит некоторую успешную ветвь SLD-дерева этой логической программы, которую обязательно находит и полный логический интерпретатор. Это означает, что при этом он находит одно из общих решений программы. Как показывает пример 14, обратное неверно: будучи применен к программе *loop*, интерпретатор Пролога впадает в бесконечный цикл.

Мы приводим несколько десятков примеров логических программ и модулей, от самых примитивных до весьма непростых, демонстрирующих основные приемы программирования на Прологе. [7] [8] Минимальной спецификацией логической прцедуры в приводимых примерах служит вводимое знаком комментария "%" описание вида:

<имя_процедуры>(
<вид_аргумента><имя_аргумента>,
...
<вид_аргумента><имя_аргумента>
): <краткое_описание_семантики_процедуры>,

[6] Фактически в этой ситуации может случиться, что в программе $\mathcal{P}$ отсутствует процедура p, и $p(\bar{t})$ есть вызов библиотечного, или как принято говорить, *встроенного* предиката. В этом случае подцель $p(\bar{t})$ устраняется из цели G_i, происходит некоторый побочный эффект, описание которого входит в определение предиката p (и излагается ниже при первом его использовании), и выполняется шаг SLD-резолюции.

[7] Все приводимые программы могут быть непосредственно исполнены в системе программирования Arity/Prolog (не младше версии 5.1). Однако за очень небольшим исключением (встроенные предикаты управления счетчиками, findall/3) эти программы написаны в ядре Пролога, и могут быть исполнены и в других системах.

[8] Многочисленные примеры программ на Прологе можно найти в книгах [6, 8].

в котором <вид_аргумента>– один из символов +, -, ?. Вид +X означает, что в момент вызова переменная X должна быть связана некоторым термом. Вид $-X$, напротив, означает, что в момент вызова переменная X должна быть свободна. Вид $?X$ не налагает на X никаких условий. Там, где это не будет вызывать недоразумений, мы будем называть интерпретатор Пролога попросту Прологом.

3.2 Первые шаги

ОТНОШЕНИЕ включения в список

```
% member(?Element,?List): Проверяет или угадывает элемент списка List
member(E,[E|_]).          % "_" -- Анонимная переменная
member(E,[_|L]) :-
  member(E,L).
```

В примере 11 мы указали два опровержения для программы, получающейся добавлением цели $:- member(2,[0,1|T])$ к этому логическому модулю. Нетрудно проверить, что первое из них является также и успешным вычислением Пролога. Так что, вызов Пролога на этой цели дает в качестве ответа решение, найденное в этом опровержении:

```
?- member(2,[0,1|T]).
T = [2|_0094] (_0094 - одна из возможных свободных переменных,
                автоматически порождаемых Прологом)
yes           (указание Пролога на успешность вычисления).
```

Второе опровержение не является вычислением Пролога, так как на третьем шаге применимо первое предложение, и потому второе применено быть не может. Между тем, нетрудно понять, что первое предложение может быть использовано только на последнем шаге вычисления, и только если в цели вида $:- member(E1,[E2|T])$ $E1$ и $E2$ оказываются унифицируемыми. До этого момента может использоваться только второе предложение, уменьшающее длину списка во втором аргументе подцели member. Отсюда следует, что любое следствие модуля Member вида $member(t1,[t2|t])$, в котором $t, t1, t2$ - термы, дает в Прологе вычислимое решение. В примере 6 мы описали минимальную модель модуля Member: множество базисных фактов вида $member(e,l)$, в которых список l содержит элемент (константу или список) e. Приведенное рассуждение показывает, что Пролог успешно достигает базисную цель вида $:- member(e,l)$. тогда и только тогда, когда атом $member(e,l)$ принадлежит наименьшей модели модуля. И все же, есть способ "заставить" Пролог найти ветвь SLD-дерева, соответствующую второму опровержению из примера 11. Для этого достаточно вызвать искусственную неудачу после того как Пролог обнаружит первую ветвь, и соответственно, первое решение. Надо в ответ на указание "yes" ввести символ ";". Это заставит Пролог выполнить возврат из последнего состояния. В нашем примере это заставит его отменить выбор первого предложения на третьем шаге, и использовать второе предложение. Тем самым, будет выполнено вычисление, соответствующее второй ветви, и получен некоторый вариант решения, указанного в примере 11:

```
?- member(2,[0,1|T]).
T = [2|_0094]
```

```
yes
--> ;
T = [_0090,2|_00A8]
yes
```

Если вновь вызвать неудачу, то получим:

```
?- member(2,[0,1|T]).
T = [2|_0094]
yes
--> ;
T = [_0090,2|_00A8]
yes
--> ;
T = [_0090,_00A4,2|_00BC]
yes
и т.д.
```

Есть и более радикальный способ: использовать встроенный предикат $fail/0$, вызов которого вызывает неудачу. Для этого надо изменить цель, например так:

```
?- member(2,[0,1|T]),write(T),nl,get0(_),fail.
```

В этой цели мы используем четыре встроенных предиката:

$write/1$, который выводит в выходной поток (в даном случае - на экран) текущее значение своего аргумента,

$get0/1$, который ожидает ввода из входного потока (в данном случае - с клавиатуры) одного символа; если аргументом является свободная переменная, то она унифицируется с соответствующим байтом (мы используем анонимную переменную "_", когда нас не интересует унификация; каждому вхождению анонимной переменной Пролог сопоставляет новую внутреннюю переменную),

$nl/0$, который начинает новую строку (точнее - пишет в выходной поток символ перевода каретки), и

$fail/0$ - предикат, вызывающий неудачу.

Эта цель вызывает бесконечный цикл: сначала Пролог проходит первую ветвь SLD-дерева, пишет на экран первое решение, переходит на новую строку, ожидает нажатия любой клавиши, затем происходит возврат. Поскольку все три встроенные подцели детерминированы, происходит возврат в точку выбора подцели $member(2,[0,1|T])$, выбирается второе предложение вместо первого, проходится вторая ветвь SLD-дерева, пишется соответствующее второе решение, и т.д. В ходе этого бесконечного цикла Пролог перебирает все успешные ветви SLD-дерева. При этом на экране мы видим нечто вроде

```
[2|_00C0]
[_00BC,2|_00D4]
[_00BC,_00D0,2|_00E8]
[_00BC,_00D0,_00E4,2|_00FC]
и т.д.
```

Мы видим, что заложенное в интерпретаторе Пролога правило перебора альтернатив и подцелей, придает рекурсивному определению предиката $member/2$ ясную операционную семантику:
Первая альтернатива: если элемент E является в списке первым, то конец вычисления; результат положителен (тело пусто, т.е. рекурсия завершается);
Вторая альтернатива: в противном случае следует отбросить первый элемент, и решать ту же задачу на более коротком списке (рекурсивный вызов $member(E, L)$).
Обратим внимание на противопоставление "в противном случае". Вторая альтернатива доступна только, если неприменима первая. Таким образом, действительно, при ее применении $E \setminus == E1$. Сравните эту операционную семантику с той логической семантикой, которая описана в примере 6.

ОТНОШЕНИЕ конкатенации списков

```
% append(?First,?Second,?Catenation): Catenation=First.Second
append([ ],L,L).
append([E|L],L1,[E|L2]) :-
  append(L,L1,L2).
```

Как и в предыдущем примере, здесь имеет место рекурсия по первому аргументу. Для цели вида $:- append(L1, L2, L)$ Пролог будет использовать второе предложение до первого момента, когда первый аргумент окажется унифицируемым с пустым списком [ ]. Заметим, что в случае, когда первый аргумент append связан определенным списком $[e1, ..., en]$, например, в цели вида $:- append([e1, ..., en], l, L)$ второе предложение используется n раз подряд, в результате чего третий аргумент будет иметь вид $[e1|[e2|...|[en|L]...]] = [e1, e2, ..., en|L]$, затем происходит вызов $append([\,], l, L)$, используется первое предложение, унифицирующее l и L, в результате чего вычисление успешно завершается с результатом $L = [e1, e2, ..., en|l]$. Иначе говоря, выполняя этот вызов, Пролог вычисляет конкатенацию списков $[e1, ..., en]$ и l. Напомним, что, как было замечено выше, наименьшая модель модуля Append состоит из базисных фактов вида $append(l1, l2, l)$, в которых $l = l1l2$. Таким образом, для целей этого вида, как и в предыдущей программе, Пролог проходит ту же ветвь SLD-дерева, что и полный интерпретатор логических программ, и соответственно, строит эквивалентное решение. Как и $member/2$, предикат $append/3$ получает ясную операционную семантику, вытекающую из правила интерпретации Пролога:
Вторая альтернатива: пока первый список непуст, его первый элемент удаляется, и помещается в конец списка-результата;
Первая альтернатива: когда же первый список пуст, и стало быть, список-результат имеет вид $[L1|L]$, где $L1$ совпадает с исходным первым списком, L отождествляется со вторым списком, т.е. выполняется конкатенация.
Отметим некоторое несовершенство этого алгоритма конкатенации: чтобы выполнить конкатенацию списков $l1$ и $l2$, Пролог выполняет $|l1| + 1$ унификацию ($|l1|$ - длина списка $l1$). В канонических языках программирования, где списки определяются указателями на рекурсивную структуру, эта операция реализуется одним присваиванием. И хотя ниже мы увидим, как можно определить конкатенацию одной унификацией (см. программу addd), соответствующее определение будет весьма неочевидным. Между тем, указанный недостаток легко превращается в преимущество: не меняя модуль, но меняя цель, можно решать и

обратную задачу, то есть уравнения типа $L\ l1\ =\ l2$ (вычисление префикса для фиксированных списков $l1, l2$) и даже $L1\ L2\ =\ l$ (перечисление всех разбиений списка l на префикс и суффикс).

```
%% Пример: префиксы, определяемые через "append":
 ?- append(Pref,_,[a,b,c,d,e]),
   write(Pref),
   tab(1),
   fail;
   true.
  [ ] [a] [a,b] [a,b,c] [a,b,c,d] [a,b,c,d,e]
  yes
```

В этой цели мы использовали два новых встроенных предиката:

$tab/1$ ($tab(k)$ пишет k символов пробела) и

$true/0$ (всегда завершающийся успехом)

и новый логический оператор

";" (при выполнении подцели $(Goal1; Goal2)$ создается точка выбора между целями $Goal1$ и $Goal2$, которые достигаются в указанном порядке; неудача $Goal1$ приводит к исполнению $Goal2$; неудача $Goal2$ вызывает неудачу всей подцели $(Goal1; Goal2)$).

Вот как происходит вычисление описанной цели:
Сначала подцель $append(Pref, [a,b,c,d,e])$ устраняется первой альтернативой Append, и потому $Pref$ сначала получает значение [ ]. $fail$ вызывает возврат в точку выбора подцели append, в результате чего выбирается вторая альтернатива, создается новая точка выбора, соответствующая подцели вида $append([a], [b,c,d,e])$, и $Pref$ получает новое значение $[a]$. Так продолжается до тех пор, пока возникает подцель вида $append([a,b,c,d,e], [\,])$, к которой неприменимы обе альтернативы, что вызывает неудачу первой ветви дизъюнкции, и затем успешное завершение второй ветви $true$.
Внимание! Если вместо конечного списка $[a,b,c,d,e]$ взять незавершенный список $[a,b,c,d,e|X]$ или попросту переменную X, то Пролог впадет в бесконечный цикл. Почему?
Еще один простой вопрос: если бы мы не предусмотрели в цели ветвь "$;true$", то вычисление завершилось бы общей неудачей "no". Почему?

Теперь уже нетрудно понять, как будет происходить вычисление Пролога, если его применить к логической программе Appr из примера 12:

```
appr(L,S,I,O):-
  append(P,S,L),
  append(P,I,O).
 ?- appr([a,b],[b],[c],R).
 R = [a,c]
 yes
```

Поскольку первые три аргумента связаны конечными списками, первый вызов append имеет вид $append(-P, +S, +L)$, а второй вызов - вид $append(+P, +I, -O)$. Действительно, в результате первого вызова переменная P связывается найденным префиксом $[a]$, затем в результате второго вызова переменная O связывается конкатенацией $[a,c]$ списков $[a]$ и

$[c]$. Тем самым, здесь Пролог проделывает то самое вычисление, дерево которого приведено на Рис. 1а) в примере 12. Важно отметить, что по завершении этого вычисления в памяти Пролога остаются две точки выбора, соответствующие двум вызовам $append$. При возврате через цель $appr([a,b],[b],[c],R)$ ближайшей является точка выбора вызова $append(P,I,O)$, и именно в нее происходит возврат. В этом состоит существеннейшая особенность языка Пролог: для каждого определения предиката надо сначала понять, как будет происходить его вызов, а затем - как будет выполняться возврат через этот вызов.

Ниже мы уже не будем разбирать вычисления Пролога с такой степенью подробности, и будем свободно оперировать уже проиллюстрированными понятиями возврата в точку выбора, возврата через вызов (т.е. подцель), перечисление решений при возвратах, и т.п.

3.3 Рекурсия, возвраты, сечение

"Угадывание" элемента списка; перечисление подсписков

```
% select(+List,?Element,?Rest): Угадывает элемент списка List и остаток Rest
select([E|L],E,L).
select([E1|L],E,[E1|L1]) :-
     select(L,E,L1).
```

Например, цель

```
?- select([a,b,c,d],E,R),write(E),tab(1),write(R),nl,fail;true.
```

заставляет Пролог перечислить в цикле возвратами все пары вида $E, R \setminus \{E\}$, в которых E - элемент списка $[a,b,c,d]$ и $R \setminus \{E\}$ - список, получающийся после его удаления:

```
 a [b,c,d]
 b [a,c,d]
 c [a,b,d]
 d [a,b,c]
 yes
```

Эта программа подобна программе Member. Разница в том, что вторая альтернатива "переписывает" в список-остаток элемент, не совпадающий с данным, а первая альтернатива вычеркивает данный элемент из результата. Чтобы понять, почему при каждом очередном возврате удаляется следующий элемент списка, надо вернуться к анализу поведения вызова member при последовательных возвратах. Предикат $select/3$ удобен для перечисления подмножеств данного списка.

```
% subrl(+List,-New): порождает без повторений посредством возвратов
                    % новые подмножества данного списка
subrl(List,List):-
  assert((sbs(List))).     % запоминает следующее подмножество в факте
                           % sbs(List)
subrl(List,New) :-
  select(List,_,Rest),
  not sbs(Rest),           % исключает повторения
  subrl(Rest,New).
```

Интерпретатор Пролога располагает динамически обновляемой базой данных, в которой хранятся факты. Для обновления базы служат некоторые встроенные предикаты. Например, использованный здесь предикат $assert/1$ при вызове вида $assert((p(a,X)))$ добавляет факт $p(a,X)$ В КОНЕЦ хранимой в базе последовательности фактов, $p(t1,u1),...,p(tk,uk)$, определяющих предикат p. Новое определение p в базе имеет вид $p(t1,u1),...,p(tk,uk)$, $p(a,X)$. Предикат $asserta/1$ ставит факт в начало последовательности, а предикат $retract/1$ при вызове $retract((p(a,X)))$ удаляет из хранимой последовательности ПЕРВЫЙ элемент $p(ti,ui)$, который унифицируем с $p(a,X)$. Эти предикаты уже не имеют простой логической семантики. Они позволяют использовать в Прологе глобальные переменные и писать программы, изменяющие себя по ходу вычисления. Встроенный предикат

$not/1$ - отрицание в Прологе - при вызове $notsbs(Rest)$ вычисляется цель $sbs(Rest)$. Если вычисление $sbs(Rest)$ завершается успешно, то подцель $notsbs(Rest)$ завершается неудачей, и наоборот, если вычисление $sbs(Rest)$ завершается неудачей, то подцель $notsbs(Rest)$ завершается успешно. Заметим, что $notsbs(Rest)$ зацикливает Пролог, если возврат в $sbs(Rest)$ вызывает цикл.

Проблемы с возвратами. Сечение

Следует констатировать, что описанный механизм управления вычислением Пролога еще не сбалансирован: - на шаге резолюции средством управления является унификация - однако имеется единственное универсальное правило возврата в случае неудачи: вернуться в ближайшую точку выбора. Это приводит к неизбежным проблемам.

```
% Пример "Программа-сваха"
% match_maker(?Person1,?Person2,?Common): находит супруга по
%                                         принципу общих интересов.
match_maker(P1,P2,Common) :-
  person(P1,m,Interests1),              % четвертая точка выбора
  person(P2,f,Interests2),              % третья точка выбора
  inter(Interests1,Interests2,Common).  % вторая и первая точки выбора
```

Увы, нет никакого способа избежать повторений !!!

```
% inter(+List1,+List2,?Common_element): находит посредством возвратов
                % все общие элементы обоих списков
inter(L1,L2,E) :-
  member(E,L1),      % вторая точка выбора
  member(E,L2).      % первая точка выбора

% База данных:
% person(?Name,?Sex,?List_of_interests).
person(basil,m,[beer,vobla,tv,girls]).
person(alex,m,[science,girls,books,sports]).
person(fedja,m,[beer,beer,beer,vobla]).
person(paul,m,[girls,cars,career,tv]).
person(leon,m,[romance,books,tv,pastry,milk]).
```

```
person(alin,f,[men,cooking,children,cats]).
person(ilin,f,[pastry,pies,meat,tv,sleeping,tv]).
person(irin,f,[sports,men,career]).
person(adel,f,[romance,men,luxury]).
person(katy,f,[men,cooking,books]).
```

Универсальным выходом из положения является:
УСТРАНЕНИЕ ТОЧЕК ВЫБОРА ПОСЛЕ ТОГО КАК НАЙДЕНО ПЕРВОЕ РЕШЕНИЕ.
Для этого используется встроенный предикат

!/0 - сечение. Областью действия предиката сечения, входящего в предложение

$$h \ :- \ b_1, ..., b_s, !, ..., b_l.$$

является часть поддерева дерева вычисления с корнем $A \quad (A = h)$, в которую входят: сам корень и все поддеревья с корнями $b_1, ..., b_s$ (см. Рис. 3).

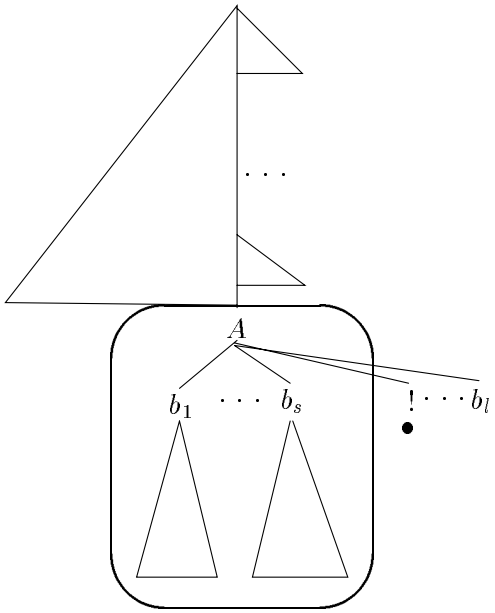


Рис.3 Область действия оператора сечения.

Эффект сечения на шаге SLD-резолюции состоит в том, что все точки выбора в области действия сечения устраняются (т.е. подцели в этой области становятся детерминированными). На шаге возврата сечение попросту вызывает неудачу. Т.о., возврат в сечение вызывает возврат в ближайшую точку выбора ВНЕ ОБЛАСТИ ДЕЙСТВИЯ СЕЧЕНИЯ. Рассмотрим следующую программу, устраняющую излишний перебор в программе *match_maker* с помощью сечения.

```
%% Модифицированная программа-сваха: в этой программе критерий отбора
%% пар усложняется: отбираются пары, имеющие хотя бы одну пару различных
%% интересов, и хоть один общий интерес. Источников избыточных повторений
%% становится еще больше, однако все они нейтрализуются сечением.
best_match_maker :-
```

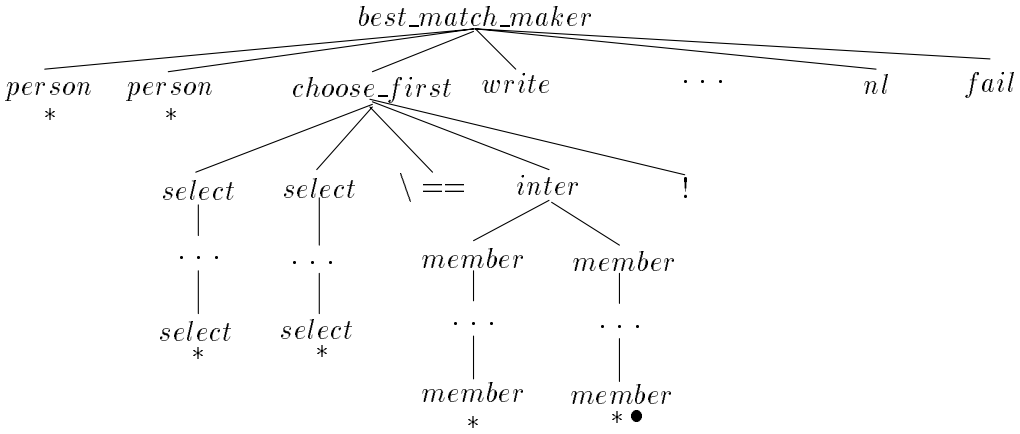
```
  person(P1,m,Int1),            % шестая точка выбора
  person(P2,f,Int2),            % пятая точка выбора
  choose_first(Int1,Int2,Comm,Diff),
  write(P1),
  write($ matches $),
  write(P2),
  write($, both like $),
  write(Comm),
  write($, differ in $),
  write(Diff),
  nl,
  fail;
  true.

choose_first(Int1,Int2,Comm,Diff) :-
  select(Int1,Diff,Rest1),     % четвертая точка выбора
  select(Int2,Diff1,Rest2),    % третья точка выбора
  Diff1 \== Diff,
% "\==" - встроенный предикат неравенства: вызов t1 \== t2 успешен,
% если значения t1,t2 не совпадают.
  inter(Rest1,Rest2,Comm),     % вторая и первая точки выбора

  !.                           % устраняет все точки выбора, возникающие
                               % в вызовах select и member; при этом точки
                               % выбора в вызовах person не задеваются
```

После этого устранения остаются две точки выбора: четвертая и третья. Следовательно, все подходящие супруги перечисляются без повторений ! На Рис. 4a) указана схема точек точек выбора непосредственно перед выполнением сечения, а на Рис. 4b) - сразу после выполнения.

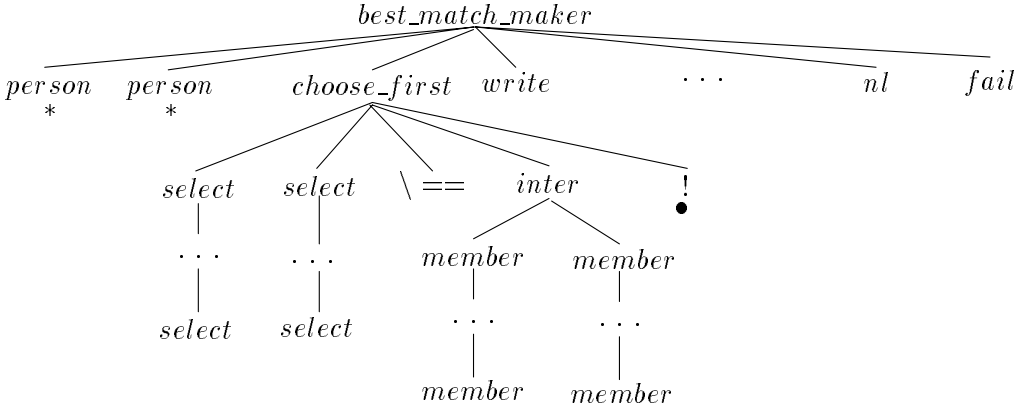
Устранение точек выбора, приводящих к некорректным результатам:

```
%% Исправление некорректного вычисления максимума двух чисел:
% max(+First,+Second,?Max): Max=maximum{First,Second}
% a) Следующее определение ошибочно:
wrongmax(F,S,F) :-  % Эта точка выбора опасна в случае возврата !
     F > S.         % встроенное арифметическое отношение.
wrongmax(_,S,S).
% Например:
:- wrongmax(3,2,M),write(M),tab(1),fail;true.
% 3 2
% yes
% Таким образом, эта точка выбора должна быть отсечена:
max(F,S,F) :-
     S < F,
```

```
    !.
max(_,S,S).
```



a) точки выбора до выполнения сечения



b) точки выбора после выполнения сечения

Рис.4 Эффект сечения.

Сечение позволяет выразить описанное выше отрицание not:

Отрицание через неудачу

```
% no(+Goal): Опровержение цели Goal (чтобы не переопределять встроенный
% в Пролог предикат $not/1,$ мы изменяем его имя).
no(Goal) :-
  Goal,!,fail.    % Опровержение
no(_).
```

Если *Goal* содержит свободные переменные, или вызовы предикатов, создающих побочные эффекты, то это определение не эквивалентно логическому отрицанию. Кроме того, вызов $no(Goal)$ может не завершиться, когда не завершается *Goal*. Важно учитывать, что отрицание через неудачу создает побочный эффект при вычислении целей со свободными переменными: его выполнение вновь делает переменные свободными.

```
% Пример:
se(X) :-
  X=a.      % "=/2" - встроенный предикат унификации; при вызове
            % t1=t2 для этого уравнения ищется НОУ
:- not((se(X),write(X)));tab(1),write(X).
% a _0054
% yes
```

Детерминизация программ

```
% concat(+Flist,+Slist,?Con): ФУНКЦИЯ конкатенации.
% Con = Flist . Slist
concat([ ],L,L) :- !.
concat([E|L],L1,[E|L2]) :-
  concat(L,L1,L2).
```

Как видим, это определение отличается от определения $append/3$ только использованием сечения в первой альтернативе. В результате, вызовы $append(t1, t2, t3)$ и $concat(t1, t2, t3)$ дают одни и те же первые решения, но возврат в $append(t1, t2, t3)$ приводит к следующему решению, тогда как возврат в $concat(t1, t2, t3)$ завершается неудачей. Другой пример - превращение отношения включения *member* в функцию проверки включения:

```
%% memb_check(+Element,+List): ФУНКЦИЯ проверки вхождения элемента в список
memb_check(E,[E|_]) :- !.
memb_check(E,[_|L]) :-
     memb_check(E,L).
```

Еще один аналогичный пример - детерминированный вариант select/3

```
%% choose(-Element,+List,-Rest): ФУНКЦИЯ поиска вхождения элемента Element
%% в список List и остатка Rest от его удаления
choose(E,[E|R],R) :- !.
choose(E,[E1|R],[E1|T]) :-
  choose(E,R,T).
```

Отметим, что детерминизация предикатов меняет ВИД ИХ ВЫЗОВА. Так, в следующем примере, не отсекая точку выбора в первой альтернативе, мы получаем ОТНОШЕНИЕ, в котором два последних аргумента имеют обший вид ?. Если же отсечь эту точку выбора, то получаем ФУНКЦИЮ, в которой третий аргумент имеет вид $-$.

```
% Отношение, позволяющее перечислять позиции и соответствующие элементы
% данного списка
% lstel(+List,?Position,?Element): отношение "Element с номером Position
% списка List"
lstel(L,I,E) :-
     le0(L,1,I,E).
le0([E|_],I,I,E).
le0([_|R],I,I2,E1) :-
     inc(I,I1),        % встроенный предикат I1 = I + 1
     le0(R,I1,I2,E1).

% Функция, вычисляющая элемент в данной позиции данного списка
% listel(+List,+Position,-Element): Element - элемент с номером Position
% списка List
listel(L,I,E) :-
     le1(L,1,I,E).
le1([E|_],I,I,E) :-
     !.
le1([_|R],I,I2,E1) :-
     inc(I,I1),
     le1(R,I1,I2,E1).
```

3.4 Методы оптимизации рекурсии и организации циклов

При реальном программировании на Прологе больших и сложных программ приходится соблюдать баланс между наглядностью логически прозрачных рекурсивных определений и удовлетворительной выполнимостью их оптимизированных вариантов. Чтобы как-то ориентироваться в затратах памяти на исполнение программы, удобно огрубленно представлять себе структуры памяти, которые использует Пролог. Их нетрудно извлечь из дерева вычисления. Простейшую структуру образует последовательность еще не устраненных к данному шагу подцелей, начинающаяся с подцели в фокусе. Эта последовательность ведет себя как обычный стек рекурсии. Фокус находится на вершине стека, и на шаге резолюции он заменяется на тело устраняющего предложения. Другая последовательность - это последовательность (в порядке возникновения) остающихся точек выбора, которая также ведет себя как стек, но на шагах возврата: при возврате ближайшая точка выбора снимается с вершины стека. Как правило, две эти последовательности либо объединяются в единый *локальный стек*, либо используются раздельные локальные стеки. Вместе с тем, требуется хранить в памяти и последовательность подстановок, определяющих связывание переменных в текущем состоянии. И эта последовательность ведет себя как стек: на шаге резолюции новый унификатор помещается на его вершину, на шаге возврата он оттуда удаляется. Этот стек (называемый *глобальным*) является аналогом кучи в стандартных компиляторах и интерпретаторах. Например, для дерева вычисления, изображенного на Рис. 1b), состояния соответствующих локальных и глобальных стеков изображены на Рис. 5.

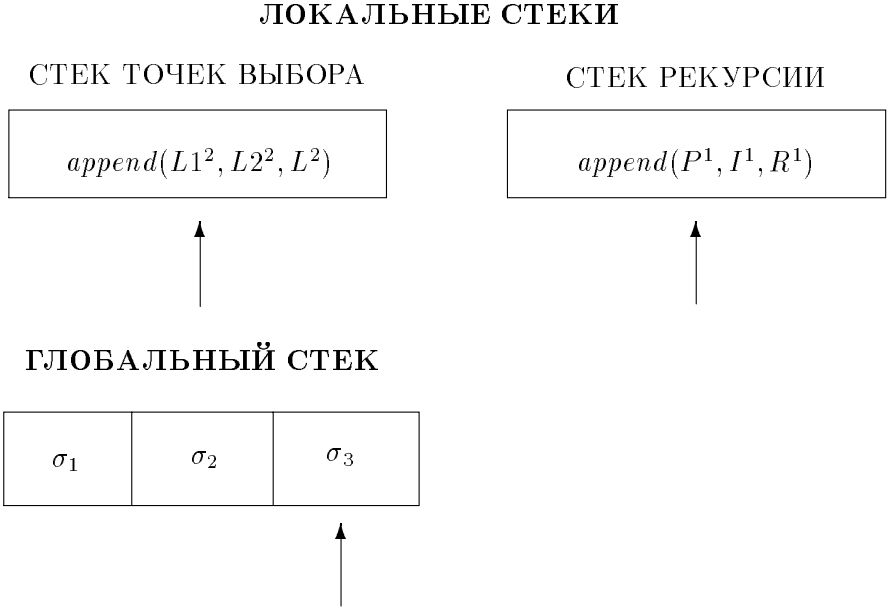


Рис.5 Состояние управляющих стеков в вычислении из примера 12

Помимо этого Пролог использует относительно небольшие стеки для синхронизации локального и глобального стеков. От размеров этих вспомогательных стеков вполне можно отвлечься. Общая схема соответствия между деревом вычисления и текущим состоянием стеков приводится на Рис. 6. Эта условная схема, конечно, не совпадает в деталях с реальной, используемой в промышленных интерпретаторах и компиляторах Пролога. Однако с точки зрения оценки затрат памяти она весьма достоверна, как показало ее сравнение с абстрактной машиной Уоррена [7] - признанным стандартом схемы компиляции Пролога.

Для Пролога выработано некоторое небольшое количество стандартных приемов оптимизации памяти (а заодно - и времени) вычислений. Наиболее употребительным методом является использование *хвостовой рекурсии.*

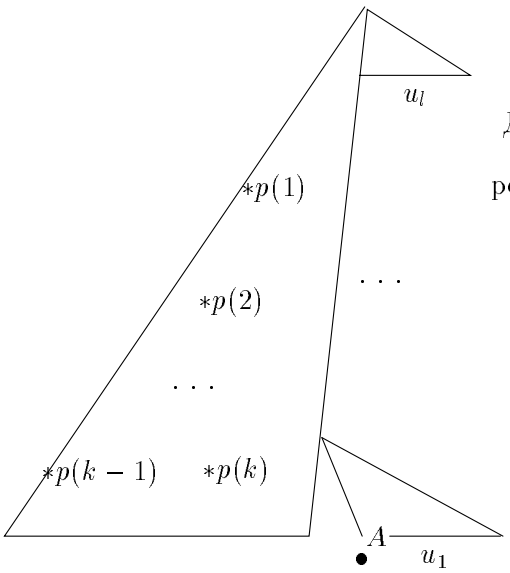
Хвостовая рекурсия:

При хвостовой рекурсии рекурсивный вызов предиката $p(\ )$ должен быть ПОСЛЕДНИМ в вычислении предиката $p(\ )$ и между последовательными рекурсивными вызовами не должны возникать точки выбора. Например, если вызов происходит в теле предложения

$$p() \ :- \ b_1, ..., b_t, p(),$$

то в момент вызова устраняемая подцель p( ) должна быть детерминирована, и дереве вычисления поддеревья с корнями $b_1, ..., b_t$ не должны содержать точек выбора. Это, в частности, означает, что хвостово-рекурсивные процедуры являются детерминированными. Нетрудно видеть, что при этом условии рекурсивный вызов помещается в стеке рекурсии всегда на одно и то же место, а стек точек выбора не растет. Таким образом, результат оптимизации хвостовой рекурсии состоит в том, что размер локальных стеков ограничен константой. Иначе говоря, хвостовая рекурсия является неявной формой итерации. Однако иногда приведение рекурсии к хвостовому виду достигается ценой увеличения кучи, т.е. глобального

стека, содержащего унифицированные и остающиеся актуальными структуры.



Дерево вычисления после n шагов

резолюции с подстановками $\sigma_1, ..., \sigma_n$

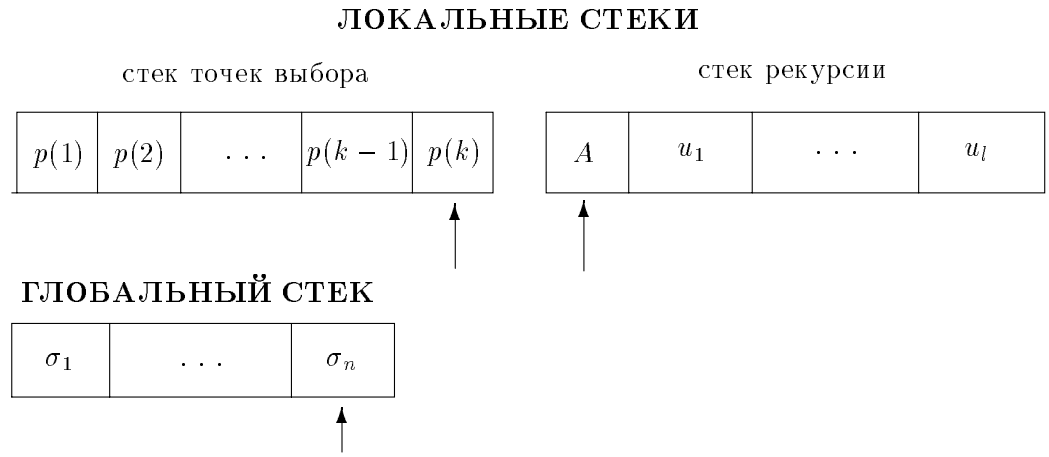


Рис.6 Схема состояния управляющих стеков.

Приведем несколько примеров.

Оптимизация зеркального отражения списков

```
% "Наивное" (не оптимизированное) зеркальное отражение списка:
% reverse(+List,?RevList): RevList - это список List в обратном порядке
nrev([ ],[ ]).
nrev([E|T],L) :-
  nrev(T,R),
  concat(R,[E],L).
```

Недостатки этой процедуры:

(1) После каждого вызова предиката *nrev* новый вызов *concat* помещается на локальный

стек, следовательно, СТЕК УВЕЛИЧИВАЕТСЯ ПРОПОРЦИОНАЛЬНО ДЛИНЕ СПИСКА и может ПЕРЕПОЛНИТЬ память, отведенную под стек.

(2) Обработка очередного элемента E списка $List$ связана с ПРОХОДОМ ЧЕРЕЗ R. Следовательно, вся процедура требует КВАДРАТИЧНОГО ВРЕМЕНИ относительно длины списка $List$.

Стандартная техника перехода к хвостовой рекурсии состоит в введении НАКОПИТЕЛЕЙ, т.е. новых вспомогательных переменных для промежуточных значений.

```
%% Хвостово-рекурсивное зеркальное отражение списка посредством введения
%% одного накопителя:

reverse(L,R) :-
  rev(L,[ ],R).          % [ ] - исходное значение накопителя
rev([ ],R,R) :- !.       % Список, являющийся вторым аргументом,
rev([E|T],A,R) :-        % накапливается в обратном порядке
  rev(T,[E|A],R).
```

Здесь В СТЕКЕ РЕКУРСИИ ИМЕЕТСЯ ДВА ЭЛЕМЕНТА (тело ЗАМЕНЯЕТ рекурсивный вызов, и параметры передаются в момент вызова), время при этом ЛИНЕЙНО !

Приведем еще один пример хвостово-рекурсивной программы, использующей накопители.

Максимальная сумма отрезка последовательности целых чисел

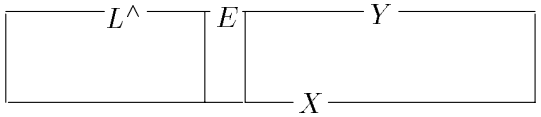
Задача: дан список целых чисел. требуется вычислить максимальную сумму чисел в отрезке этого списка. Например, в последовательности $[-7, 2, -1, 9, -5, 6, -10]$ эта сумма равна 11, а соответсвующим отрезком является список $[2, -1, 9, -5, 6]$. Алгоритм решения состоит в том, что, просматривая список слева-направо без возвратов, следует помнить два значения: сумму текущего отрезка S и абсолютный максимум суммы просмотренных отрезков Ms. Если $S > 0$, то следующий элемент прибавляется к сумме текущего отрезка (если нужно, то при этом увеличивается Ms), в противном случае следующий элемент начинает новый отрезок.

```
% maxsum(+List_of_ints,-Maxsum).
maxsum([ ],0) :- !.
maxsum([E|R],Ms) :-
  mxs(R,E,E,Ms).

% mxs(+List,+Sum_of_interv,+Absol_Max,-Max)
mxs([ ],S,Ms,M) :-        % M - окончательный результат
  !,
  max(S,Ms,M).
mxs([E|L],S,Ms,M) :-
  S > 0,                  % отрезок будет продолжен
  !,
```

```
  Ns is E+S,
  max(Ns,Ms,Nm),
  mxs(L,Ns,Nm,M).
mxs([E|L],_,Ms,M) :-     % S =< 0, E начинает новый отрезок
  max(E,Ms,Nm),
  mxs(L,E,Nm,M).
```

Другая техника перехода к хвостовой рекурсии состоит в введении так называемых ДИФФЕРЕНЦИАЛЬНЫХ СПИСКОВ.

```
% mirr(+List,?RevList): RevList - это список List в обратном порядке
% RevList = List^. Например, [b,a]^ = [a,b].
mirr(List,Rev) :-
  mir(List,Rev-[ ]).       % Второй аргумент является дифференциальным
                           % списком, задающим простой список Rev
mir([ ],Rev-Rev) :- !.     % Rev-Rev задает пустой список [ ]

mir([E|L],X-Y) :-          %
  mir(L,X-[E|Y]).          % см. Рис.7
```



$$X - Y = L^\wedge + E \Longrightarrow X - [E|Y] = L^\wedge$$

Рис.7. Схема дифференциальных списков во втором предложении *mir*

Дифференциальные списки интересны тем, что их конкатенация требует единственной унификации. Тем самым, например, можно за счет одной унификации поместить элемент в конец списка, что невозможно сделать, используя предикат append/3.

```
% Конкатенация дифференциальных списков
conc(L-R,R-T,L-T).

% Добавление элемента в конец дифференциального списка;
% результат - дифференциальный список
addd(L-X,E,C-Y) :-
  conc(L-X,[E|Z]-Z,C-Y).
% L-X=L-R, [E|Z]-Z=R-T, C-Y=L-T означает, что если PX=L и
% QY=C, то Q=PE.

% Добавление элемента в конец дифференциального списка;
% результат - список
add(L-X,E,C) :-
  conc(L-X,[E|Z]-Z,C-[ ]).
```

```
% Пузырьковая сортировка. "Наивное" определение с квадратичным временем.
% bsort(+InList,?SortedList):
bsort([ ],[ ]) :- !.
bsort([E|T],S) :-
  bsort(T,S0),
  posit(E,S0,S).  % помещает E в надлежащую позицию в S0, результат - S

% posit(+Element,+InList,?ResList): Element "всплывает" в списке InList и
          % перемещается до надлежащей позиции в соответствии
          % с отношением порядка leq
posit(E,[ ],[E]) :- !.
posit(E,[E1|T],[E,E1|T]) :-    % надлежащая позиция найдена
  leq(E,E1),
  !.
posit(E,[E1|T],[E1|R]) :-    % еще не найдена, "всплывание" продолжается
  posit(E,T,R).

% Отношение порядка:
leq(E1,E2) :-
  E1 @=< E2.          % Здесь leq/2 - лексикографический порядок термов
                      % @=< можно заменить любым другим отношением порядка

%% BINSORT - самая быстрая сортировка: слиянием (за время N*logN)
%% Алгоритм: Список разбивается примерно пополам, каждая часть
%%           сортируется (рекурсивным применением той же процедуры)
%%           по-отдельности, затем отсортированные списки сливаются
%%           в один отсортированный списох-результат.
% msort(+List,?Sorted_list).
msort([ ],[ ]) :-
  !.
msort([E],[E]) :-
  !.
msort(L,S) :-
  part(L,L1,L2),     % разбиение L на две части РАВНОЙ ДЛИНЫ: L1,L2
  msort(L1,S1),      % сортировка первой части
  msort(L2,S2),      % сортировка второй части
  merge(S1,S2,S).    % слияние отсортированных списков в один посредством
                     % помещения элементов на правильные позиции.
% Сделано только log |L| разбиений / слияний
% part(+List,?Left_half,?Right_half): List = Left_half . Right_half
          % Длины списков Left_half и Right_half отличаются максимум
          % на единицу
part(L,L1,L2) :-
  length(L,N),
```

```
  M is N // 2,
% is/2 - встроенный предикат, представляющий присваивание: X is e
% унифицирует со свободной переменной X значение выражения e. Если
% при вычислении значения возникает ошибка, то e получает значение ?.
% Встроенный предикат "//" выполняет деление целых чисел (результатом
% является целое число)
  sect(L,M,L1,L2).   % L1 - начало списка L длины M; L2 - остаток списка.

% sect(+List,+Prefix_length,?Prefix,?Rest): List = Prefix . Rest
                     % | Prefix | = Prefix_length
sect(L,0,[ ],L) :- !.
sect([E|T],N,[E|R],S) :-
  dec(N,N1),
  sect(T,N1,R,S).

% merge(+Sorted1,+Sorted2,?Merged_and_Sorted).
                          % например, merge([1,3,6],[2,4],[1,2,3,4,6])
                          % хвостово-рекурсивное определение:
merge([ ],L,L) :- !.
merge(L,[ ],L) :- !.
merge([E|T1],[E|T2],[E|T]) :-        % первые элементы совпадают
  !,
  merge(T1,[E|T2],T).
merge([E1|T1],[E2|T2],[E1|T]) :-     % первые элементы различны и
  lt(E1,E2),                         % E1 < E2
  !,
  merge(T1,[E2|T2],T).
merge([E1|T1],[E2|T2],[E2|T]) :-     % первые элементы различны и
  lt(E2,E1),                         % E1 > E2
  merge([E1|T1],T2,T).
```

Структурные операторы циклов возвратами

Другой распространенный прием оптимизации памяти - это использование циклов возвратами. Мы уже видели примеры возвратов в цикле.

```
% forall(+Goal,+Cond): Для всех решений цели Goal выполняется
%                       условие Cond
forall(C,L) :-
  not((C,not(L))).

%% Пример: список состоит из четных чисел
?- forall(member(N,[2,6,44,108]),N mod 2 == 0).
% "==/2" - встроенный предикат совпадения значений, mod - встроенный
% арифметический оператор "остаток от деления нацело"
```

```
% for(+Left,+Right,-Num): перечисляет посредством возвратов целые
  % числа Num в интервале [Left,Right]; аналог цикла FOR в Паскале.
for(L,L,L) :- !.
for(L,R,L) :-           % присваивает Num = Left
   L =< R.
for(L,R,N) :-           % перемещает Left вправо
   L < R,
   L1 is L+1,
   for(L1,R,N).

%% Пример:
?- for(1,10,N),write(N),tab(1),fail;true.
 1 2 3 4 5 6 7 8 9 10
 yes
```

Неструктурные циклы repeat

repeat/0 - один из важнейших, и одновременно простейших встроенных предикатов, служащих для организации циклов возвратами. Он имеет определение:

```
repeat.
repeat :-
  repeat.
```

В этом определении мы немедленно узнаем аналогичное определение предиката $r/0$ из примера 14. Вызов *repeat* всегда успешен, и создает точку выбора. Возврат в эту точку приводит к использованию второй альтернативы, новому вызову, и возобновлению точки выбора. Важно, что на это практически не тратится память.

```
%% Пример цикла repeat:
%% text_file_len(+File,-Num_of_lines): Число строк в ACCII файле
text_file_len(File,N) :-
  open(H,File,r),          % Открывает файл File для чтения, возвращает
                           % системный указатель H файла File
  ctr_is(0,C0),            % запоминает значение счетчика 0 в переменной C0
  ctr_set(0,0),            % счетчику 0 присваивает значение 0
  repeat,                  % точка выбора для итерации
  (read_line(H,_),         % читает одну строку из файла H
   ctr_inc(0,_),           % увеличивает на 1 значение счетчика 0
   fail;
   close(H),               % закрывает файл H
   ctr_is(0,N),            % берет результат из счетчика 0
   ctr_set(0,C0)           % восстанавливает значение счетчика 0 из C0
  ).
```

Приведем пример программы, использующей разнообразные циклы возвратами.

Обучающаяся программа, распознающая простоту натуральных чисел

Чем больше числа, простота которых проверяется, тем больше чисел, простота которых распознается программой таблично. Динамическая таблица простых чисел хранится в базе данных. Она содержит факты $pn(P)$ для начального отрезка простых чисел и факт $pm(M)$, где M - наибольшее число в таблице. Для данного числа N сначала проверяется, нет ли его в таблице простых чисел. Если его нет, то проверяется, имеется ли в таблице число, большее квадратного корня R из N (т.е.$M > R$). Если имеется, то простота N проверяется на основании таблицы (там находятся все простые делители составных чисел меньших $N + 1$). В противном случае таблица расширяется.

```
prim :-
      consult(knowprim),!,     % Таблица начального отрезка можества
                               % простых чисел в файле knowprim.ari
                               % загружается в базу данных Пролога
                               % встроенным предикатом concult/1
      proc.
prim :-
      assert((pm(3))),         % Таблицы нет; создаем в БД двухэлементную
      assert((pn(2) :- !)),    % таблицу
      asserta((pn(3) :- !)),
      proc.

proc :-
    repeat,                    % точка входа в цикл
    write($Input a natural number followed by '.'> $),
    read(N),                   % встроенный предикат, читающий терм N из
                               % входного потока
    ifthenelse(p(N),
               (nl,write(N),write($ is prime$)),
               (nl,write(N),write($ is composite$))),
% ifthenelse(Cond,Branch1,Branch2) - встроенный предикат Arity/Prolog'a.
% Если цель Cond завершается успешно, то затем выполняется Branch1,
% если - неудачно, то выполняется Branch2. Возврат происходит в ту
% цель, которая была выполнена. Возврат в Cond завершается неудачей
% вызова ifthenelse. Имеется и аналогичный предикат ifthen/2, и
% оператор case (см. ниже)
    pm(P),     % P - максимальное число в таблице
    nl,
    write($my greatest prime is: $),
    write(P),nl,
    ask.
ask :-         % Продолжить сеанс ?
    write($>more?(y/n)$),
```

```
    nl,
    write(>),
    get(A),
    case([(A==‘y;A==‘Y)      % в Arity ‘y - байт, соответсвующий
                             % символу "y" (в ascii: 121)
            -> (nl,fail),       % продолжение: возврат в начало цикла
          not(keys(new))        % конец; таблица не изменена
            -> true|            % ветвь, выполняемая, если предыдущие
                                 % ветви завершились неудачей
                                 % конец; расширение таблицы
          (file_list(knowprim,[pn/1,pm/1]),
     % встроенный предикат, сохраняющий в файл содержащиеся в БД
     % определения процедур, имена которых перечислены в списке, поверх
     % старых определений
           abolish(pn/1),        % встроенный предикат, удаляющий из БД
           abolish(pm/1),        % все определение предиката
           abolish(new/0))
          ]).

% p(+Number): проверка Number на простоту
p(N) :-
    isqrt(N,R),                 % R - корень из N
    inc(R,R1),
    pm(M),
    ((N=<M,!,pn(N));            % N имеется в таблице
     (R1=<M,!,prn(N,R1));       % простые делители, не превышающие
                                % корня из  N имеются в таблице
     gp(N,R1,M)),!.             % надо продолжить таблицу

% prn(+N,+R): проверка некратности N простым числам в интервале [2,R]
prn(N,R) :-
    forall((for(2,R,D),pn(D)),
            N mod D =\= 0).

% gp(+N,+R,+M): проверка некратности N простым числам в интервале [2,R],
% сопровождаемая порождением новых простых чисел в интервале (M,R+1).
gp(N,R,M) :-
    newp(P,M),          % порождение в цикле новых простых чисел, не
                        % превышающих R+1
    P >= R,!,           % конец цикла
    assert((new)),      % флажок "таблица расширена"
    prn(N,R).

% newp(-NewNum,+Num): NewNum - простое число, следующее за Num
newp(P,M) :-
```

```
    M1 is M+2,
    newn(M1,P),     % цикл для подбора следующего простого числа P
    isqrt(P,R),
    inc(R,R1),      % встроенный предикат "R1 = R+1"
    prn(P,R1),      % конец цикла
    retract((pm(X))),  % изменение таблицы в БД
    assert((pm(P))),
    asserta((pn(P) :- !)).

% Порождение следующего натурального числа
newn(S,S).
newn(S,N) :-
    inc(S,S1),
    newn(S1,N).

% isqrt(+N,-R): R - целая часть корня квадратного из N
isqrt(N,R) :-
   for(2,N,R),
   (R+1)^2 > N,!.
```

Циклы возвратами оптимальнее хвостовой рекурсии по отношению к памяти, так как при возвратах чистятся все стеки Пролога, однако они требуют доступа к глобальным переменным (значениями счетчиков, значениями, хранимыми в фактах БД, и т.д.), что связано с определенной дополнительной затратой времени.

4 Специфические задачи и средства

Применение Пролога для решения специфических задач нередко связано с использованием специфических встроенных предикатов и даже приемов программирования. Поэтому мы приведем несколько весьма упрощенных программ, иллюстрирующих использование специальных средств.

4.1 Лексический и синтаксический анализ

Лексический анализатор

Дан СПИСОК байтов. Задача состоит в проверке его лексической корректности (по отношению к словарю) и преобразовании его в соответствующий список лексем; не содержащиеся в словаре лексемы помечаются символом #.

```
% lexan(+Chars,-/+Words).
lexan([Char|Rest],[Word|Words]) :-
  word(Rest,Char,Word,Tail),    % Word - начало списка [Char|Rest], Tail -
                                % его конец
  rest_stream(Tail,Words).
```

```
rest_stream([13],[ ]) :-
  !.
rest_stream([Char|Chars],[Word|Words]) :-
  word(Chars,Char,Word,Rest),
  rest_stream(Rest,Words).

% word(+Chars,+Start_char,-/+Word,-/+Rest_of_chars).Распознает следующее слово
word(Chars,Char,Word,Chars) :-
  unit_word(Char),               % Слово из одного символа
  !,
  list_text([Char],Word).
% list_text(?List_of_bytes,?String): встроенный предикат, преобразующий
% строки в списки байтов, и наоборот. Например,
% list_text([119,111,114,100],word).
word(Chars,Char,Word,Left_chars) :-
  lex_char(Char,Low_Char),       % Строчные или заглавные буквы
  !,
  rest_word(Chars,Rest,Left_chars),
  (dict(Word,[Low_Char|Rest]),  % слово найдено в словаре
   !;
   list_text([35,Low_Char|Rest],Word)).  % 35=‘# помечает ненайденную лексему
word([Char|Chars],_,Word,Rest) :-        % пробелы
  word(Chars,Char,Word,Rest).

rest_word([Char|Chars],[LCh|Rest],Left_chars) :-
  lex_char(Char,LCh),
  !,
  rest_word(Chars,Rest,Left_chars).
rest_word(Tail,[ ],Tail).

unit_word(44) :- !.  % ,
unit_word(39) :- !.  % ;
unit_word(58) :- !.  % :
unit_word(63) :- !.  % ?
unit_word(33) :- !.  % !
unit_word(46) :- !.  % .

lex_char(Ch,Ch) :-      % a,b,c,..
  96 < Ch,
  Ch < 123,
  !.
lex_char(Ch,Low) :-     % A,B,C,... преобразуем в  a,b,c,..
  64 < Ch,
  Ch < 91,
```

```
  !,
  Low is Ch+32.
lex_char(Ch,Ch) :-     % 1,2,...,9
  47 < Ch,
  Ch < 58,
  !.
lex_char(39,39) :-    % '
  !.
lex_char(45,45) :-    % -
  !.
lex_char(0,0).        % 0

%% СЛОВАРЬ %%

% dict(?Word_or_digit,?List_of_bytes)
dict(apple,[97,112,112,108,101]) :- !.
dict(an,[97,110]) :- !.
dict(gave,[103,97,118,101]) :- !.
dict(i,[105]) :- !.
dict(him,[104,105,109]) :- !.
dict(Digit,[N]) :-
  (47 < N,
   N < 58
   ;
   N==0),
  !,
  list_text([N],Digit).
```

Синтаксический анализатор

Дан СПИСОК лексем. Задача состоит в проверке его синтаксической корректности и построении его синтаксической структуры - в виде дерева составляющих.

"Наивный" вариант

Дерево составляющих *Struct* представляются термом. Лежащая в основе анализатора идея крайне проста: проверяя интервал входного списка на соответствие некоторой синтаксической категории (например, категории *sentence*) мы перечисляем всевозможные его декомпозиции на два смежных подинтервала, и для каждого из них проверяем, что он относится к определенной синтаксической категории (например, первый подинтервал - к категории *nounphrase*, а второй - к категории *verbphrase*). Декомпозиции определяются предикатом append, содержащим точку выбора. Поэтому при неудаче проверки возврат в точку выбора приводит к новой декомпозиции.

```
% sent(+Sentence,-Struct).
```

```
sent(S,st(X,Y)) :-
  append(Np,Vp,S),  % декомпозиции S на Np,Vp при возвратах в эту
                    % точку выбора
  nphrase(Np,X),
  vphrase(Vp,Y).

nphrase(Np,np(X,Y)) :-
  append(D,Dp,Np),
  determ(D,X),
  dnphrase(Dp,Y).

determ([D],det(D)) :-
  dict_ry(D,determ).

dnphrase(Dp, dphr(X,Y)) :-
  append(Adj,Np,Dp),
  adject(Adj,X),
  dnphrase(Np,Y).

dnphrase([N],n(N)) :-
  dict_ry(N,nn).

adject([A],adj(A)) :-
  dict_ry(A,aj).

vphrase(Vp,vp(X,Y)) :-
  append(V,Np,Vp),
  verb(V,X),
  nphrase(Np,Y).

verb([V],v(V)) :-
  dict_ry(V,vb).

%% СЛОВАРЬ %%

% dict_ry(?Lexem,?Synt_category): словарь.
dict_ry(the,determ) :- !.
dict_ry(a,determ) :- !.
dict_ry(an,determ) :- !.
dict_ry(boy,nn) :- !.
dict_ry(ball,nn) :- !.
dict_ry(girl,nn) :- !.
dict_ry(apple,nn) :- !.
dict_ry(silly,aj) :- !.
dict_ry(tall,aj) :- !.
```

```
dict_ry(fat,aj) :- !.
dict_ry(kicks,vb) :- !.
dict_ry(likes,vb) :- !.
dict_ry(eats,vb).
```

Версия с дифференциальными списками

```
synal(S,st(X,Y)) :-
  nphr(S-Vp,X),            % (ср. с определением конкатенации
  vphr(Vp,Y).              %  дифференциальных списков)

nphr(Ph-R,np(X,Y)) :-
  dterm(Ph-T,X),
  dnphr(T-R,Y).

dterm([D|R]-R,det(D)) :-    % одноэлементный дифференциальный список
  dict_ry(D,determ),!.      % позиция лексемы "привязывается" к системе
                            % составляющих
dnphr([N|R]-R,n(N)) :-
  dict_ry(N,nn),!.

dnphr(Ph-R, dphr(X,Y)) :-
  adjct(Ph-T,X),
  dnphr(T-R,Y).

adjct([A|R]-R,adj(A)) :-
  dict_ry(A,aj),!.

vphr(Vp,vp(X,Y)) :-
  vrb(Vp-R,X),
  nphr(R-[ ],Y).

vrb([V|R]-R,v(V)) :-
  dict_ry(V,vb),!.

?- sent([a,fat,silly,boy,eats,an,apple],S).
 S=st(np(det(a),dphr(adj(fat),dphr(adj(silly),n(boy)))),
      vp(v(eats),np(det(an),n(apple))))
 yes

?- synal([a,fat,silly,boy,eats,an,apple],S).
% S=st(np(det(a),dphr(adj(fat),dphr(adj(silly),n(boy)))),
%      vp(v(eats),np(det(an),n(apple))))
% yes
```

Эти две версии синтаксического анализатора несравнимы по эффективности. Первая наивная версия осуществляет полный перебор всех возможных разбиений составляющих на непосредственные составляющие, и для каждого варианта проверяет его приемлемость. Второй анализатор осуществляет анализ без возвратов целенаправленным поиском унифицируемых вариантов.

Логические грамматики (ЛГ)

В стандартном Прологе имеется замечательная возможность пользоваться встроенным анализатором. Вместо того чтобы писать анализирующую программу, вы пишете бесконтекстную грамматику, порождающую анализируемый язык. Для этого имеется очень простой формат записи правил грамматики. При загрузке грамматики в БД Пролог транслирует ее в эффективный анализатор с дифференциальными списками. Теперь, чтобы выполнить анализ, достатолчно вызвать предикат, описание которого однозначно определяется грамматикой: если s - аксиома (т.е. начальный нетерминал) грамматики, то вызов анализатора имеет вид:

$$s(-P1, ..., -Pk, +List\_to\_analyse, [\,]).$$

Здесь список $List\_to\_analyse$ - это и есть анализируемая фраза (список лексем). Параметры $P1, ..., Pk$ вы выбираете сами. Дело в том, что в отличие от бесконтекстных грамматик ЛГ имеют параметры-переменные, смысл которых аналогичен смыслу прологовских локальных переменных. Так что, и ваш начальный нетерминал может иметь свои параметры. Вот они-то и входят в список $P1, ..., Pk$. Стандартный параметр начального символа - строящаяся по ходу анализа синтаксическая структура, задаваемая, как и в предыдущих примерах, термом. Еще одним достоинством логических грамматик является возможность помещать в правые части правил любые прологовские цели. Они вводятся конструкцией $Goal$. Это позволяет выполнять по ходу анализа любые программы на Прологе, передавая им параметры нетерминалов, что создает неограниченные возможности для выполнения в ходе синтаксического анализа семантических процедур.

Мы не станем точно описывать синтаксис логических грамматик. Вместо этого мы приведем пару простых примеров и сделаем следующее замечание:

1) Аналогом терминалов в ЛГ являются выражения вида $[X]$, смысл которых: прочесть из входного списка очередную лексему X.

2) Каждый нетерминал описывает процедуру анализа для описываемых им составляющих. Порядок вызова этих процедур, конечно же, тот же, что и в Прологе: правила и символы в их правых частях перебираются анализатором в указанном порядке. Отсюда и те же предосторожности - надо следить, чтобы при анализе не возник бесконечный цикл (думайте о нетерминале как о предикате!). Итак, приведем примеры.

ЛГ, эквивалентная анализатору с дифференциальными списками

```
synal(s(X,Y)) -->           % Единственный параметр аксиомы -
                            % восстанавливаемая система составляющих
  nphr(X),
  vphr(Y).
nphr(np(X,Y)) -->
  dterm(X),
```

```
  dnphr(Y).
dterm(determ(X)) -->
  {X = a;X = the;X = an},   % т.е. присвоить X либо a, либо - the, либо an
                            % возникает две точки выбора
  [X].                      % прочесть из входного списка X
dnphr(np(X,Y)) -->
  adj(X),
  dnphr(Y).
dnphr(X) -->
  noun(X).
adj(aj(X)) -->
  {X = silly;X = tall;X = fat},
  [X].
noun(n(X)) -->
  {X = boy;X = ball;X = girl;X = apple},
  [X].
vphr(vp(X,Y)) -->
  verb(X),
  nphr(Y).
verb(v(X)) -->
  {X = kicks;X = likes;X = eats},
  [X].

ex(S) :-
  synal(S,[a,fat,silly,boy,kicks,an,apple],[]).

?- ex(S).
 S=st(np(det(a),dphr(adj(fat),dphr(adj(silly),n(boy)))),
      vp(v(kicks),np(det(an),n(apple))))
 yes
```

ЛГ, порождающая формулы химических соединений

```
% Мы используем стандартный синтаксис химических формул, с той лишь
% особенностью, что индексы пишутся через двоеточие, и формулы
% оканчиваются символом ";". Пример формулы: "$Al:2(SO:4):3;$" .

ch_form -->
  [';];             % конец формулы
  el_form,
  ch_form.
el_form -->
  sign;
  sign,
  [':],            % начало индекса
```

```
  ind;
  [`(],           % начало группы в скобках
  brack_form,
  [`:],
  ind.
brack_form -->   % конец группы в скобках
  [`)];
  el_form,
  brack_form.
sign -->
  [Cap],          % Первая буква - заглавная
  {64 < Cap,      % цель, проверяющая ограничения на буквы
   Cap < 91},
  sign_rest.
sign_rest -->
  [ ];             % знак элемента без индекса
  [B],
  {96 < B,        % прочие буквы знака элемента - прописные
   B < 123},
  sign_rest.
ind -->
  [D],            % цифра
  {47 < D,
   D < 58}.

?- ch_form("Al:2(SO:4):3;",[ ]).
 yes
?- ch_form("Al:2O:3;",[ ]).
 yes
?- ch_form("Ag:2O:3;",[ ]).
 yes
```

4.2 Структуры как данные; символические вычисления

Поверхностный синтаксис бинарных операторов

В Прологе можно вводить собственные операторы, используемые в выражениях как бинарные функторы с заданными ПРЕДШЕСТВОВАНИЕМ и АССОЦИАТИВНОСТЬЮ. Встроенный предикат

$$op(+Precedence, +Assoc, +Functor)$$

определяет новый оператор; встроенный предикат

$$current\_op(?Prec, ?Assoc, ?Func),$$

перечисляет операторы при возвратах.
ПРЕДШЕСТВОВАНИЕ: $1 \leq number \leq 1200$. Чем больше предшествование оператора, тем слабее он связывает:

$?-current\_op(400, yfx, *), current\_op(500, yfx, +)$. означает, что внутренним представлением выражения $X + Y * Z$ является $+(X, *(Y, Z))$.
АССОЦИАТИВНОСТЬ: задает порядок операторов и позицию функтора по отношению к его аргументам для операторов с одинаковым предшествованием.
1. Для унарных операторов:

```
fx - неассоциативный префикс ("case","?-");
fy - префикс ("not","+"(sign));
yf - постфикс.
```

2. Для бинарных операторов: xfx - инфиксный, неассоциативный (":-");

```
xfy - право-ассоциативный (current_op(1000,xfy,','))
```

Следовательно, внутренним представлением для (X, Y, Z) является $(X, (Y, Z))$.

```
yfx - лево-ассоциативный ('+','*')
```

Следовательно, внутренним представлением выражения $X + Y + Z$ является $((X + Y) + Z)$.

Приведем пример введения новых операторов.

Символическое вычисление пропозициональных формул

```
:-op(670,xfy,&).     % вводим оператор конъюнкции,
:-op(675,xfy,or).    % дизъюнкции,
:-op(900,fy,#).      % классического отрицания

p.  % Эта пропозициональная буква определяется как истинная;
    % все остальные считаются ложными.

sat(X) :- X.
sat(true).
sat(X & Y) :-        % не будь приведенного выше определения &,
                     % в этом месте Пролог выдал бы синтаксическую
                     % ошибку
  sat(X),
  sat(Y).
sat(X or Y) :-
  sat(X).
sat(X or Y) :-
  sat(Y).
sat(# X) :-
  unsat(X).
sat(# X) :-
  not(X).

unsat(X or Y) :-
```

```
  unsat(X),
  unsat(Y).
unsat(X & Y) :-
  unsat(X).
unsat(X & Y) :-
  unsat(Y).
unsat(# X) :-
  sat(X).

?- sat(#(p&(#p))). % да
 yes

?- sat(p&(#p)).    % нет
 no

?- unsat(p&(#p)).  % да
 yes
```

Древесные грамматики или системы переписывания термов

```
% Бесконтекстные древесные грамматики с правилами вида "tree1 --> tree2"
% tree_comp(+Tree,?Path,?ResultTree,+Subtree,+Substitut): применяет
% правило Subtree-->Substitut к дереву Tree в вершине, идентифицируемой
% путем Path (См. Рис. 8). В результате получается дерево ResultTree.
% При возвратах перечисляет все способы применения правила к дереву Tree.
% Это преобразование может в равной степени интерпретироваться и как
% замена вхождения унифицируемого подтерма данного терма на некоторый терм.

tree_comp(T,P,T1,S,S1) :-
  tc(T,P-P,T1,S,S1).

tc(T,P-[ ],S1,S,S1) :-
  T=S.           % подстановка и наследование остаточных поддеревьев
tc(T,P-D,T1,S,S1) :-
  functor(T,F,N),
% встроенный предикат, который для данного терма T=f(u1,...,uk)
% унифицирует F с f, и N с k, или же для заданных F=f и N=k
% унифицирует T с f(_,...,_).
  for(1,N,I),
  arg(I,T,TI),
% встроенный предикат, который для данного терма T=f(u1,...,uk)
% и даного номера I=i унифицирует TI с ui.
  functor(TI,F0,_),
  argrep(T,I,New,T1),
% встроенный предикат, который для данного терма T=f(u1,...,ui,...,uk),
```

```
% даного номера I=i, и данного терма New унифицирует T1 с
% f(u1,...,New,...,uk).
  addd(P-D,F0,P1-D1),       % Добавление элемента в конец
                            % дифференциального списка (см. выше)
  tc(TI,P1-D1,New,S,S1).

deriv_step(T1,T2,N,P) :-    % непосредственная выводимость (за один шаг)
  t_gram(N,S1,S2),          % угадывает правило (точка выбора)
  tree_comp(T1,P,T2,S1,S2).

% deriv(+From,-To,+Steps): Выводит дерево TO из дерева From за Steps шагов.
deriv(T1,T1,0) :- !.
deriv(T1,T2,N) :-
  deriv_step(T1,T,_,_),
  N1 is N-1,
  deriv(T,T2,N1).

% drv(+From,-To): Выводит To из From в интерактивном режиме
drv(T1,T2) :-
  write($MORE?(y/n)>$),
  get0(X),
  nl,
  drv_proc(X,T1,T2).

drv_proc(X,T1,T2) :-
  X \= `y,
  write($done$).
drv_proc(X,T1,T2) :-
  deriv_step(T1,T,_,_),
  write(T),
  nl,
  drv(T,T2).

% Пример древесной грамматики:
t_gram(1,g(a,X),h(X,X)).
t_gram(2,a(b,X),a(nb,X,X)).
% Примеры выводов:
?- tree_comp(
      a(c(j),f(1,g(a,k(e)),n(0),g(A,j(1,2,3))),d),
      P,
      T1,
      g(a,X),
      h(X,X)
          ).
```

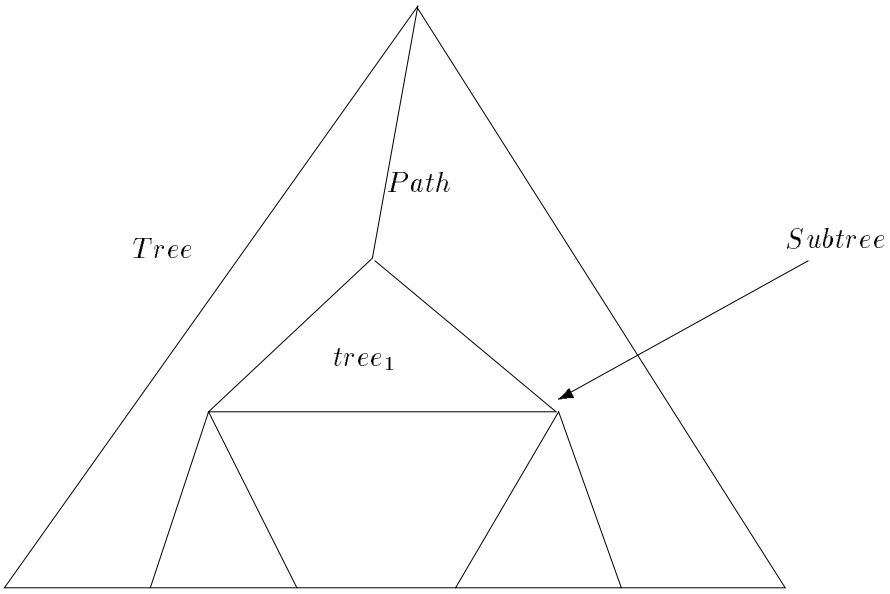


Рис.8. Подстановка вместо поддерева.

```
?- deriv_step(a(c(j),f(1,g(a,k(e))),d),U,N,P),
   write(U),tab(1),write(N),tab(1),write(P),
   nl,
   fail;
   write($all decompositions$).
?- drv(a(c(j),f(1,g(a,k(e)),n(0),g(A,j(1,2,3))),d),T).
```

Синтез программ

Чтобы проиллюстрировать технику автоматического синтеза программ на Прологе, мы выберем следующий, хотя и упрощенный, но все же достаточно представительный пример. Допустим, требуется синтезировать ЦЕЛЬ Пролога, вычисляющую условное арифметическое выражение, исходя из описания алгоритма его вычисления на некотором языке спецификации, включающем присваивание, условный оператор типа if-then-else, и обычные арифметические операторы (см. ниже примеры спецификаций и синтеза). Представим себе далее, что в результате фазы синтаксического анализа спецификация транслируется в списки $[op1, op2, op3, ...]$, элементами которых являются либо безусловные команды вида:

$asn(v(X), v(Y))$, (где X и Y - переменные, X - свободна, и $v(X), v(Y)$ - их внутренние представления), соответствующая присваиванию $X := Y$,

$pl(v(X), v(Y), v(Z))$, соответствующая $Z = X + Y$,

$sb(v(X), v(Y), v(Z))$, соответствующая $Z = X - Y$,

$ml(v(X), v(Y), v(Z))$, соответствующая $Z = X * Y$,

$dv(v(X), v(Y), v(Z))$, соответствующая $Z = X//Y$,

либо условная команда вида:

$con(C,T,F)$, соответствующая $ifthenelse(C1,T1,F1)$, где:
T и F - списки операторов, являющиеся результатом трансляции соответственно $T1$ и $F1$, и C - результат трансляции элементарного условия - имеет вид $v(X)\omega v(Y)$, где ω - одно из стандартных арифметических бинарных отношений: $=$, $<$, $\leq$, ... Наконец, условимся, что в языке спецификации имеются *входные*, *выходные* и *вспомогательные* переменные, имеющие соответственно внутренние представления:

$v(X_n) = [i,n]$ - $n-$ая входная переменная,

$v(Z_n) = [o,n]$ - $n-$ая выходная переменная,

$v(A_n) = [v,n]$ - $n-$ая вспомогательная переменная, и (для простоты) имеется всего 10 вспомогательных переменных.

При этих предположениях описываемая ниже программа $Synt$, синтезирующая прологовскую цель $Goal$, вычисляющую выражение, специфицируемое списком операторов Ops с Iv входными переменными и Ov выходными переменными, создает предложение вида

$$calc(X_1,...,X_{Iv},Z_1,...,Z_{Ov}) \ :- \ Goal.$$

(с автоматически сгенерированными прологовскими переменными) и помещает его в файл $prog.ari$. Сгенерированную таким образом программу $calc$ можно загрузить в БД Пролога, и выполнить на тех или иных значениях входных переменных.

```
%% synt(+Ops,+Iv,+Ov): синтезирует программу вычисления последовательности
%%                  операторов Ops с Iv входными и Ov выходными переменными.
synt(Ops,Iv,Ov) :-
  gener(Iv,Invl),           % порождает список прологовских переменных
  gener(Ov,Ouvl),           % см. ниже
  vars(Varl),               % факт БД, хранящий список вспомогательных
                            % переменных
  concat(Invl,Ouvl,Parml),  % функция конкатенации списков (см. выше)
  syn(Ops,Invl,Ouvl,Varl,Body), % собственно процедура синтеза цели Body
  create(H,'prog.ari'),
% вызов встроенного предиката create(H,'prog.ari') создает новый файл
% prog.ari и внутренний указатель на него H
  Head =.. [calc|Parml],
% вызов встроенного предиката "=.." вида X =..[f,a1,...,ak] унифицирует
% СВОБОДНУЮ переменную X с термом f(a1,...ak). Его вызов вида
% f(a1,...,ak) =.. Y унифицирует H с [f,a1,...,ak]
  write(H,(Head :- Body)),
% пишет в файл сгенерированное предложение как ТЕРМ. Формат записи терма t
% в файл с исходным текстом: t "." Поэтому в файл пишется ".":
  write(H,'.'),
  close(H).                 % файл закрывается

vars([X1,X2,X3,X4,X5,X6,X7,X8,X9,X10]).  % факт, хранящий список свободных
                                         % вспомогательных переменных

new(X). % При вызове этого факта Пролог создает НОВУЮ свободную переменную!
```

```
gener(0,[ ]) :-!.       % Генерация списка N разных свободных переменных
gener(N,[V|T]) :-
  N > 0,
  dec(N,N1),           % Встроенный предикат N1 = N - 1
  new(V),
  gener(N1,T).

% syn(+Ops,+Inputs,+Outputs,+Auxvars,-Goal): Ops - последовательность
% операторов, Inputs, Outputs, Auxvars - количество входных, выходных
% и вспомогательных переменных, Goal - синтезируемая цель
syn([ ],_,_,_,true) :- !.  % Концу списка операторов соответствует
                           % тождественно истинная цель true
% Синтез условного оператора
syn([cnd(C,T,F)|Ro],Invl,Ouvl,Varl,(ifthenelse(C1,T1,F1),Rb)) :-
  !,
  ctran(C,Invl,Ouvl,Varl,C1),  % преобразование условия (см. ниже)
  syn(T,Invl,Ouvl,Varl,T1),
  syn(F,Invl,Ouvl,Varl,F1),
  syn(Ro,Invl,Ouvl,Varl,Rb).

% Синтез оператора присваивания
syn([asn(X,Y)|Ro],Invl,Ouvl,Varl,(Subgl,Rb)) :-
  !,
  tran(X,Invl,Ouvl,Varl,Tx),   % синтез прологовских переменных (см. ниже)
  tran(Y,Invl,Ouvl,Varl,Ty),
  Subgl = (var(Tx),!,Tx=Ty;
% вызов встроенного предиката var(X) успешен т.и т.т., когда X -
% свободная переменная
           write('''error: '''),
% пишет в файл строку 'error: '
           write(X),write(''' is bound'''), fail
           ),
  syn(Ro,Invl,Ouvl,Varl,Rb).

% Синтез оператора +
syn([pl(X,Y,Z)|Ro],Invl,Ouvl,Varl,(Tz is Tx+Ty,Rb)) :-
  !,
  tran(X,Invl,Ouvl,Varl,Tx),
  tran(Y,Invl,Ouvl,Varl,Ty),
  tran(Z,Invl,Ouvl,Varl,Tz),
  syn(Ro,Invl,Ouvl,Varl,Rb).

% Синтез оператора -
syn([sb(X,Y,Z)|Ro],Invl,Ouvl,Varl,(Tz is Tx-Ty,Rb)) :-
  !,
```

```
  tran(X,Invl,Ouvl,Varl,Tx),
  tran(Y,Invl,Ouvl,Varl,Ty),
  tran(Z,Invl,Ouvl,Varl,Tz),
  syn(Ro,Invl,Ouvl,Varl,Rb).

% Синтез оператора деления нацело //
syn([dv(X,Y,Z)|Ro],Invl,Ouvl,Varl,(Tz is Tx//Ty,Rb)) :-
  !,
  tran(X,Invl,Ouvl,Varl,Tx),
  tran(Y,Invl,Ouvl,Varl,Ty),
  tran(Z,Invl,Ouvl,Varl,Tz),
  syn(Ro,Invl,Ouvl,Varl,Rb).

% Синтез оператора умножения  *
syn([ml(X,Y,Z)|Ro],Invl,Ouvl,Varl,(Tz is Tx*Ty,Rb)) :-
  !,
  tran(X,Invl,Ouvl,Varl,Tx),
  tran(Y,Invl,Ouvl,Varl,Ty),
  tran(Z,Invl,Ouvl,Varl,Tz),
  syn(Ro,Invl,Ouvl,Varl,Rb).

% Генерация переменных в операторах и условиях
syn([Ex|Ro],Invl,Ouvl,Varl,(Tex,Rb)) :-
  ctran(Ex,Invl,Ouvl,Varl,Tex),
  syn(Ro,Invl,Ouvl,Varl,Rb).

ctran(Ex,Invl,Ouvl,Varl,Nex) :-
  Ex =.. [O,A1,A2],
  Nex =.. [O,T1,T2],
  tran(A1,Invl,Ouvl,Varl,T1),
  tran(A2,Invl,Ouvl,Varl,T2).

tran(I,_,_,_,I) :-
  (integer(I);I==err),
  !.
% вызов встороенного предиката integer(I) успешен т. и т.т., когда
% I - целое число. константа err - стандартный результат ошибки
% вычисления выражения
tran([i,I],Invl,_,_,X) :-
  !,
  listel(Invl,I,X).  % функция, вычисляющая I-ый элемент списка (см. выше)
tran([o,I],_,Ouvl,_,X) :-
  !,
  listel(Ouvl,I,X).
tran([v,I],_,_,Varl,X) :-
```

```
  !,
  listel(Varl,I,X).
tran(Ex,Invl,Ouvl,Varl,Nex) :-
  ctran(Ex,Invl,Ouvl,Varl,Nex).
```

Вот два примера синтеза:

```
% Цель
ex1 :-
  synt([
pl([i,1],[i,2],[v,1]),
ml([v,1],4,[v,2]),
sb([v,2],[i,3],[v,3]),
cnd([i,1] > 0,
    [dv([v,3],[i,1],[o,1])],
    [[o,1] = err])
      ],3,1).
% приводит к синтезу программы:
calc(_02D0,_0338,_03A0,_040C) :-
  _0478 is _02D0 + _0338 ,
  _0480 is _0478 * 4 ,
  _0488 is _0480 - _03A0 ,
  ifthenelse(_02D0 > 0,
    (_040C is _0488 // _02D0 , true),
    (_040C = err , true)) ,
  true.
% которую можно вызвать, например, так:
?- calc(1,2,3,Z).
 Z = 9
 yes

% Другая цель:
ex2 :-
  synt([
pl([i,1],[i,2],[v,1]),
ml([v,1],4,[v,2]),
sb([v,2],[i,3],[v,3]),
       asn([v,3],[i,1])
      ],3,1).
% требует синтезировать программу для выражения, в котором происходит
% присваивание связанной переменной. Соответствующий результат:
calc(_0264,_02CC,_0334,_03A0) :-
  _040C is _0264 + _02CC ,
  _0414 is _040C * 4 ,
  _041C is _0414 - _0334 ,
  (var(_041C) , ! , _041C = _0264 ;
```

```
   write('error: ') ,
   write([v,3]) ,
   write(' is bound') ,
   fail) ,
  true.

% вызвав эту программу, получаем:
?-calc(1,2,3,Z).
 error: [v,3] is bound
 no
```

Метаинтерпретация

Внимательный читатель, разбирая программу $Synt$, должен был задуматься над тем, почему в вызове $write(H,(Head:-Body))$ выражение $(Head:-Body)$ является термом. Такое возможно только если :- и ',' являются инфиксными бинарными операторами. Это нетрудно проверить, используя встроенный предикат $display/1$:

```
?- display((p(X,Y,Z) :- a,b,c)).
 ':-'(p(_0038,_004C,0060),','(a,','(b,c)))
 yes
```

Итак, в самом деле, ':-' и ',' - инфиксные операторы. ':-' имеет (в Arity-прологе) предшествование 1200 (т.е. связывает слабее всех операторов) и ассоциативность xfx (единственное инфиксное вхождение), а ',' имеет предшествование 1000 (слабее него лишь дизъюнкция ';' (1100) и ':-') и ассоциативность xfy (инфиксный, право-ассоциативный). Для операторов этого типа Пролог опускает очевидные группирующиеся направо скобки при внешнем изображении термов. Поэтому терм $',',(a,',',(b,c))$ имеет внешнее представление a,b,c. Таким образом, всякая цель, всякое предложение и тело этого предложения - суть термы, а программа - не что иное как последовательность термов. Эта замечательная особенность Пролога (роднящая его с Лиспом) является основанием для элегантной техники программирования метапрограмм, которые применяясь к цели некоторой программы Пролога $\mathcal{P}$, и эмулируют вычисление Пролога. Эта техника называется *метапрограммированием*. Ее смысл в том, что, эмулируя вычисление Пролога, можно вызывать в нужных точках те или иные процедуры (именно так пишутся, например, программы-отладчики), и даже изменять стандартную стратегию. Идея метапрограммирования элементарна. Вот простейший метаинтерпретатор:

```
metaint(true) :- !.           % конец цели
metaint((Subgoal,Goal)) :-    % интерпретация составной цели
  !,
  metaint(Subgoal),           % сначала - интерпретация первой подцели
  metaint(Goal).              % затем - интерпретация остатка цели
% Важное замечание: Сечение ПРЕДШЕСТВУЕТ рекурсивным вызовам metaint,
% поэтому точки выбора для целей Subgoal и Goal сохраняются.
metaint(Subgoal) :-           % интерпретация вызова предиката
  clause(Subgoal,Body),       % точка выбора интерпретируемой программы
```

```
  metaint(Body).
% Вызов встроенного предиката clause(Subgoal,Body) первое предложение
% Head :- Body, для которого Head = Body. Clause создает точку выбора,
% в которой число альтернатив совпадает с количеством предложений в
% определении предиката подцели Subgoal. При возврате clause возвращает
% следующее унифицируемое предложение. Таким образом, можно в цикле
% возвратами перечислить все унифицируемые предложения.
```

Метапрограммирование связано с некоторыми специфическими проблемами. одна из них состоит в том, что во избежание коллизии определений предикатов следует разделить программу и метапрограмму. Например, метапрограмма компилируется, а программа интерпретируется (находится в БД Пролога). В Arity-прологе имеется дополнительная возможность поместить метапрограмму и программу в разные миры БД и интерпретировать и ту, и другую. Другая проблема состоит в том, что Пролог ведет собственное внутреннее управление переменными, и потому в случаях, когда метаинтерпретатор должен манипулировать переменными и их промежуточными значениями, он не должен производить побочный эффект на переменные Пролога, и следовательно, должен работать с КОПИЯМИ прологовских переменных. Мы проиллюстрируем эту своеобразную технику на примере упрощенного метаинтерпретатора, выводящего сообщения об успешности / неуспешности вызовов подцелей и о промежуточных значениях переменных. При этом мы отвлечемся от первой проблемы, считая, что в интерпретируемой программе попросту не используются предикаты, определяемые в метаинтерпретаторе. Кроме того, мы максимально упростим синтаксис интерпретируемых программ, не допуская в них операторов управления типа ifthenelse, case, и т.д., и - главное - оператора сечения, метаинтерпретация которого представляет серьезную проблему, которую мы обсудим отдельно.

```
% Метаинтерпретатор ядра Пролога (без сечения), визуализирующий
% состояния промежуточных подцелей.
% run(+Goal): Goal - интерпретируемая цель.
run(Goal) :-
  write($GOAL TO ACHIEVE> ?- $),
  newgoal(s,Goal,SGoal,_,Surf),
% поверхностное представление копии подцели Goal
  write(SGoal),
  newgoal(i,Goal,IGoal,Vars,_),
% внутреннее представление копии подцели Goal с переменными
  ifthenelse(meta(IGoal,Vars,Surf),
       (nl,                          % вызов метаинтерпретатора успешен
        write($The goal ?-$),
        write(SGoal),
        write('.'),
        write($ has succeeded$)),
       (nl,                          % вызов метаинтерпретатора неудачен
        write($The goal ?-$),
        write(SGoal),
        write('.'),
```

```
        write($ has failed$))
            ).

% Создание поверхностной (Flag = s) или внутренней (Flag = i) копии
% цели Goal.
% newgoal(+Flag,+Goal,-NGoal,-Vars,-Surf): NGoal - копия цели Goal. Vars -
% внутренние переменные метаинтерпретатора, Surf - их внешние обозначения.
newgoal(Flag,Goal,NGoal,Vars,Surf) :-
  freeze(Goal,FG,Nv),               % переменные "замораживаются" т.е.
                                    % связываются константами
  genvs(Nv,Vars,Surf),              % порождаются переменные и их обозначения
  melt(Flag,FG,Vars,Surf,NGoal).    % Переменные цели FG восстанавливаются

% freeze(+Term,-Frozen_term,-Number_of_vars): Frozen_term - копия терма Term,
% в которой переменные заменены константами вида '$var'(N), где N - номер
% переменной. Number_of_vars - число переменных в Term.

freeze(T,FT,N) :-
  new_copy(T,FT),                   % FT - копия T с новыми переменными
  numvars(FT,N).                    % процедура замораживания

% new_copy(+Term,-New): New - копия Term с новыми переменными
new_copy(Term,New) :-
  assert((renew(Term))),
  retract(renew(New)).

% процедура замораживания
% numvars(+Term,-N): N - число различных переменных в Term
numvars(Term,N) :-
  numvar(0,Term,N).

% numvar(+N1,+Term,-N2): N1 - число переменных, предшествующих анализируемому
% вхождению Term, N2 - число переменных по завершении просмотра Term
numvar(N1,Term,N2) :-
  nonvar(Term),                 % не переменная
  !,
  functor(Term,F,A),
  numv(0,A,N1,Term,N2).         % цикл по подтермам
numvar(N,'$var'(N1),N1) :-      % !! Связывание ВСЕХ вхождений переменной
  N1 is N+1.

numv(A,A,N,_,N) :-              % конец цикла по подтермам
  !.
numv(I,A,N1,Term,N3) :-         % шаг цикла по подтермам
  I < A,
```

```
  I1 is I+1,
  arg(I1,Term,Arg),
  numvar(N1,Arg,N2),
  numv(I1,A,N2,Term,N3).

% genvs(+N,-Var,-Surf): порождает списки Var и Surf N переменных
% и их обозначений
genvs(N,Var,Surf) :-
  N > 0,
  gnvs(0,N,Var,Surf).

gnvs(N,N,[ ],[ ]) :- !.
gnvs(I,N,[V|T],[Sa|S]) :-
  inc(I,I1),
  new(V),                       % вызов факта new(V) со свободной
  int_text(I1,NI),              % переменной V порождает новую переменную
% вызов встроенного предиката, преобразующего целое число I1 в
% представляющую его строку NI
  concat($X$,NI,Sf),
  atom_string(Sa,Sf),
% вызов встроенного предиката, преобразующего строку Sf в атом Sa
  gnvs(I1,N,T,S).

new(X).                         % факт БД

% melt(+Flag,+Frozen,+Vars,+Surf,-Term): при Flag = i Term - это терм,
% получаемый из "замороженного" терма Frozen заменой кодов переменных
% на соответствующие переменные из Vars; при Flag = s Term - это терм,
% получаемый из терма Frozen заменой кодов переменных на соответствующие
% их обозначения из Surf.
melt(F,'$var'(N),Vars,Surf,Var) :-  % терм - код переменной
  !,
  ifthenelse(F==i,                % восстановление переменных
        listel(Vars,N,Var),       % см. выше определение listel/3
        listel(Surf,N,Var)).      % замена констант обозначениями переменных
melt(_,X,_,_,X) :-                % терм - константа
  atomic(X),!.
% вызов встроенного предиката atomic(X) успешен т. и т.т., когда X -
% переменная или константа; в данном случае - константа
melt(F,FT,Vars,Surf,MT) :-        % составной терм
  functor(FT,Fn,A),
  functor(MT,Fn,A),
  melta(F,A,FT,Vars,Surf,MT).

melta(F,A,FT,Vars,Surf,MT) :-
```

```
  mlta(F,0,A,FT,Vars,Surf,MT).    % цикл по подтермам

mlta(_,A,A,_,_,_,_) :- !.         % конец цикла по подтермам
mlta(F,I,A,FT,Vars,Surf,MT) :-    % шаг цикла по подтермам
  inc(I,I1),
  arg(I1,FT,FA),
  arg(I1,MT,MA),
  melt(F,FA,Vars,Surf,MA),
  mlta(F,I1,A,FT,Vars,Surf,MT).

% Собственно метаинтерпретатор, применяемый к "размороженной" цели
% meta(+Goal,+Vars,+Surf): Goal - интерпретируемая цель, Vars - ее
% переменные, Surf - их обозначения.
meta(true,_,_) :- !.               % конец цели
meta((Sub,Goal),Vars,Surf) :-      % интерпретация составной цели
  !,
  meta(Sub,Vars,Surf),
  meta(Goal,Vars,Surf).
meta(Sub,_,_) :-                   % подцель - свободная переменная. Неудача
  var(Sub),
  !,fail.
meta(Sub,_,_) :-                   % подцель - вызов встроенного предиката
  functor(Sub,P,A),
  Pred=..[/,P,A],
  system(Pred),
% вызов встроенного предиката system(P/A) успешен т. и т.т., когда P/A -
% встроенный предикат
  !,
  Sub.
meta(Sub,Vars,Surf) :-             % подцель - вызов определяемого предиката
 clause(Sub,Body),
 nl,
 write($SUBGOAL> $),
 write(Sub),
 outvars(Vars,Surf),               % сообщение о значениях переменных
 wait_key,                         % ожидает нажатия клавиши
 meta(Body,Vars,Surf);             % вызов неудачен
 nl,
 write($SUBGOAL> $),
 write(Sub),
 outvars(Vars,Surf),
 write($ fails !$),
 wait_key,
 fail.
```

```
% outvars(+Vars,+Surf): выводит сообщение о значениях переменных
outvars([ ],[ ]) :- !.
outvars([V|T],[S|R]) :-
  nl,write(S),
  ifthenelse(var(V),
       write($ is free$),
       (write(=),write(V))),
  tab(1),
  outvars(T,R).

% wait_key ожидает нажатия клавиши
wait_key :-
  wa(1,116), % встроенный предикат, окрашивающий позицию курсора в цвет 116
  put(31),
% встроенный предикат, выводящий в позицию курсора символ с ASCII кодом 31
  keyb(_,_).
% keyb(?A,?S) - встроенный предикат при вызове которого из входного
% потока ожидается некоторый символ; A унифицируется с ascii-кодом,
% S - со scan-кодом символа (DOS-зависимый предикат).
```

Чтобы проиллюстрировать применение метаинтерпретатора *run*, рассмотрим программу, состоящую из синонимичных определений предикатов $member/2$ и $append/3$:

```
mbm(E,[E|_]).
mbm(E,[_|T]) :-
  mbm(E,T).

ap([ ],L,L).
ap([E|L1],L2,[E|L]) :-
  ap(L1,L2,L).
?- mbm(E,[1,2]),ap([E},[3,4],[ ]).
```

Применение *run* к цели этой программы вызовет следующий поток сообщений:

```
?- run((mbm(E,[1,2]),ap([E},[3,4],[ ]))).
 GOAL TO ACHIEVE> ?- mbm(X1,[1,2]),ap([X1},[3,4],[ ])
 SUBGOAL> mbm(1,[1,2])
 X1 = 1
 SUBGOAL> ap([1},[3,4],[ ])
 X1 = 1 fails!
 SUBGOAL> mbm(_3DF8,[1,2])
 X1 is free
 SUBGOAL> mbm(2,[1,2])
 X1 = 2
 SUBGOAL> ap([2},[3,4],[ ])
 X1 = 2 fails!
 SUBGOAL> mbm(_3DF8,[2])
```

```
X1 is free
SUBGOAL> mbm(_3DF8,[ ])
X1 is free fails!
SUBGOAL> mbm(_3DF8,[2])
X1 is free fails!
SUBGOAL> mbm(_3DF8,[1,2])
X1 is free fails!
The goal ?- mbm(X1,[1,2]),ap([X1},[3,4],[ ]). has failed
yes
```

Теперь уже совсем нетрудно обогащать синтаксис интерпретируемых программ условными и другими операторами, вводя в определение предиката meta/3 дополнительные альтернативы. Например, чтобы интерпретировать ";" и $ifthen/2$, следует добавить предложения:

```
meta((G1;G2),Vars,Surf) :-
  !,
  (meta(G1,Vars,Surf);
   meta(G2,Vars,Surf)).
meta(ifthen(Cond,G),Vars,Surf) :-
  (meta(Cond,Vars,Surf),
   !,
   meta(G,Vars,Surf);
  true).
```

Теперь вернемся к проблеме метаинтерпретации сечения. Может показаться, что его интерпретирует альтернатива

```
meta(Sub,_,_) :-
  functor(Sub,P,A),
  Pred=..[/,P,A],
  system(Pred),
  !,
  Sub.
```

поскольку !/0 - встроенный предикат. Однако, это определение неверно оределяет область действия оператора сечения. Корректное определение - значительно более тонкое - содержит следующий метаинтерпретатор $Metacut$.

Метаинтерпретатор ядра Пролога Metacut

```
% metacut(+Goal,-CUT):
% Шаг резолюции через сечение "!" порождает ближайшую точку выбора
% (ситуация 1), но при этом оставляет свободной переменную-флажок CUT.
% Возврат в созданную точку выбора не вызывает неудачу интерпретации, но
% связывает переменную CUT. Таким образом, после возврата в "!" выполняется
% ШАГ РЕЗОЛЮЦИИ, А НЕ ШАГ ВОЗВРАТА (что, строго говоря, некорректно).
% Однако, перейдя к интерпретации подцелей тела, следующих за "!",
```

```
% metacut ("чувствуя", что флажок CUT связан), не интерпретируя эти
% подцели просто достигает конца тела (ситуация 3).
% При интерпретации конца тела возможны две ситуации:
% (ситуация 4) выход происходит при связанной переменной CUT, что означает,
% что уже произошел возврат в сечение в теле предложения C. В этом случае
% происходит неудачное завершение подцели, которая была устранена
% предложением C. Это и устраняет некорректность порядка вычисления.
% (ситуация 5) выход из тела происходит либо, если оно не содержит
% сечения, либо если сечение есть, но в него не было возврата. В этом
% случае переменная-флажок CUT свободна.
metacut(_,CUT):- nonvar(CUT),!.    % Ситуация 3: флажок связан, вызов подцели
                                   % завершается успешно, без исполнения
% Таким образом, для нормального исполнения требуется, чтобы CUT не был связан
metacut((!),CUT):- true;           % Ситуация 1: шаг резолюции через "!",
                                   % возникает точка выбора и CUT не связан
                   CUT=(!),        % Ситуация 2: возврат в "!" не приводит
                   !.              % к неудаче, но связывает CUT
metacut((Sub,Goal),CUT) :- !,
                     metacut(Sub,CUT),
                     metacut(Goal,CUT).
metacut((Goal1;Goal2),CUT) :- !,
                     (metacut(Goal1,CUT);
                      metacut(Goal2,CUT)
                      ).
metacut(not(Goal),_) :- !,
                     not(
                         (metacut(Goal,CUT),var(CUT))
                         ),!.
metacut(call(Goal),CUT) :- !,
                     metacut(Goal,CUT).
metacut(findall(T,Cond,List),CUT) :- !,
                     findall(T,
                             (metacut(Cond,CUT),
                              (nonvar(CUT),!,fail;true)
                              ),
                             List).
% встроенный предикат, вызов которого вида findall(+T,+Cond,-List)
% унифицирует List со списком всех частных случаев терма T при
% подстановках, являющихся решениями цели Cond.
metacut(fail,_) :- !,fail.
metacut(true,_) :- !.
metacut(Sub,_) :-
               functor(Sub,P,A),
               (system(P/A),!,
                call(Sub);
```

```
                clause(Sub,Body),
                metacut(Body,CUT),
                (nonvar(CUT),
                 !,fail   % ситуация 4: неудача на выходе из отсеченного
                 ;        % тела моделирует возврат в сечение
                 true)    % ситуация 5: выход из тела либо, если оно не
                          % отсечено, либо отсечено, но до возврата в "!"
                ).
```

Можно проверить, как работает этот интерпретатор, сверив выполнение следующих целей:

```
ex1 :-
  match_maker(P1,P2,Comm),
  write(P1),
  write($ matches $),
  write(P2),
  write($, both like $),
  write(Comm),
  nl,
  fail;
  true.

tst1 :-
  metacut(ex1,CUT).

ex2 :-
  match_maker1(P1,P2,Comm,D),
  write(P1),
  write($ matches $),
  write(P2),
  write($, both like $),
  write(Comm),
  write($, differ in $),
  write(D),
  nl,
  fail;
  true.

tst2 :-
  metacut(ex2,CUT).

ex3 :-
  person(P1,m,Int1),
  person(P2,f,Int2),
  choose_first(Int1,Int2,Comm,Diff),
  write(P1),
```

```
  write($ matches $),
  write(P2),
  write($, both like $),
  write(Comm),
  write($, differ in $),
  write(Diff),
  nl,
  fail;
  true.

tst3 :-
  metacut(ex3,CUT).
```

Следует отметить, что метапрограммирование связано с большими затратами памяти и времени. Дело в том, что естественные определения метаинтерпретаторов не являются хвостово-рекурсивными, и потому, при метаинтерпретации пропадает эффект оптимизации рекурсии в интерпретируемой программе. В принципе возможен полностью итеративный метаинтерпретатор. Однако он неизбежно будет использовать глобальные переменные в БД Пролога, и потому будет тратить чрезмерно много времени на обмен с БД.

4.3 Вычисления на графах

Раскраска плоского графа в 3 цвета

Задача: дан плоский неориентированный граф. Требуется проверить, можно ли так раскрасить вершины графа в три цвета, чтобы смежные вершины были окрашены в разные цвета. Эта задача решается [9] классическим для Пролога методом перебора, который называется *перечисление-фильтрация*. Идея метода описывается простой схемой: $p(X), f(X)$. Генератор решений $p(X)$ содержит точку выбора, при возвратах в которую перечисляются всевозможные значения X. Неудача вызова фильтра f на очередном значении X вызывает новый возврат в точку выбора, и так до тех пор, пока f не окажется истинным, и цикл завершится. В задаче раскраски роль генератора p играет предикат, перечисляющий всевозможные раскраски, а роль фильтра играет предикат, выражающий нарушение условия корректности раскраски.

```
A. Сначала покажем, как выглядит решение для графа с заданным
% множеством вершин. Граф задается дугами-фактами вида adj(V1,V2).
% Пример описания графа с пятью вершинами {a,b,c,d,e}:
adj(a,b).
adj(b,c).
adj(d,e).
adj(c,d).
adj(e,a).
adj(a,d).
```

[9] Напомним, что любой плоский неориентированный граф можно раскрасить в четыре цвета, но это в общем случае неверно для трех цветов.

```
adj(b,e).

% Ребро:
aj(X,Y) :-
  adj(X,Y);adj(Y,X).

% cl3_5(-L): L - функция раскраски пяти вершин.
cl3_5(L) :-
  L=[[a,C1],[b,C2],[c,C3],[d,C4],[e,C5]],
  ch(C1),ch(C2),ch(C3),ch(C4),ch(C5),
  not cond(L).

% ch(-C): Выбор цвета C
ch(C) :-
  C=1;C=2;C=3.

% Условие некорректности раскраски
cond(L) :-
   member([V1,C],L),
   aj(V1,V2),
   member([V2,C],L).

% B. Теперь продемонстрируем решение в общем случае, когда число
% вершин графа неизвестно заранее.

% Перечисление всевозможных 3-раскрасок графа, задаваемого
% дугами-фактами вида adj(V1,V2).
cl3 :-
  col_form(L),    % построение списка-формы L функции раскраски
  guess_col(L),   % угадывание раскраски
  not cond(L),    % проверка корректности раскраски
  show(L),        % перечисление всех решений
  fail;
  true.

% Построение списка-формы функции раскраски.
% col_form(-L): L - форма функции раскраски графа.
col_form(L) :-
  findall([V,C],
          (adj(V,_);adj(_,V)),
          Lst),
  rptless(Lst,L).

% Устранение повторений:
% rptless(+List,-Ens): Ens - список без повторений элементов List
```

```
rptless(List,Ens) :-
  rl(List,[ ],Ens).
rl([ ],L,L) :- !.
rl([[V,_]|R],Acc,L) :-
  mck(V,Acc),      % тест на повторение подтвердился
  !,
  rl(R,Acc,L).
rl([[V,C]|R],Acc,L) :-
  rl(R,[[V,C]|Acc],L).

% Тест на повторение:
mck(E,[[E,_]|_]) :- !.
mck(E,[_|R]) :-
  mck(E,R).

% Угадывание раскраски.
guess_col([ ]) :- !.
guess_col([[_,C]|R]) :-
  ch(C),
  guess_col(R).

% Демонстрация решения.
show([ ]) :-
  !,nl.
show([[V,C]|R]) :-
 write(cl(V)),write($=$),write(C),write($, $),
 show(R).
```

Кратчайший путь в графе с весами

Задача: дан неориентированный граф, дуги которого помечены натуральными числами (см., например, граф на Рис. 9, изображающий фрагмент карты железных дорог). Для данной пары различных вершин - входной и выходной - требуется найти путь из первой во вторую с минимальным весом (вес пути - сумма весов его дуг). Задача решается методом "прямой волны": последовательно строится множество вершин, достижимых из входной вершины $From$ вдоль путей без повторений вершин, имеющих L дуг, для $L = 1, 2, ...$ до тех пор, пока в этом множестве не окажется выходная вершина To. По ходу построения подсчитываются длины путей (суммы весов дуг), и выбирается путь наименьшей длины.

```
% path(+From,+To,-Length). m - минимальный путь, любая другая клавиша -
% очередной кратчайший путь. В этой программе мы иллюстрируем еще один
% способ описания переменных: идентификатор, начинающийся с символа "_"
path(_f, _t, _L) :-
  ctr_is(0,C0),
  ifthen(clause(found(_f,_t,_),_),  % результат предыдущего запроса
         abolish(found/3)),
```

```
  assert((found(_f,_t,[ ]))), % found/3 накапливает найденные пути из _f в _t
  write(
$"m" - minimal path; any other key - next shortest path through backtracking.$
        ),
  nl,
  write($Type "m" or any other key > $),
  ifthenelse((keyb(109,_),cls),           % `m = 109
             m([[0, [_f]]], _t),
             (ctr_set(0,0),
              fn([[0, [_f]]], _f, _t),   % любая другая клавиша
              ctr_set(0,C0))),
  report(_f,_t,_L),
  abolish(found/3).
```

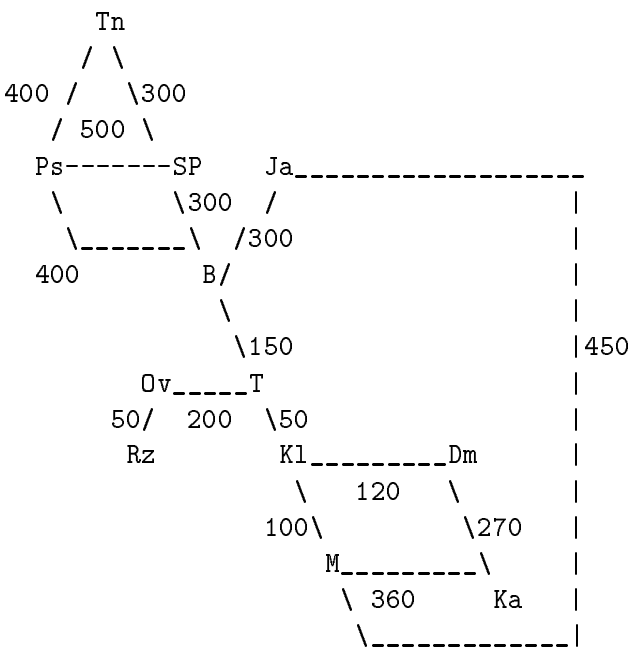


Рис.9. Фрагмент карты железных дорог

```
% Железнодорожная база данных (см. Рис.9)
% d(?Point,?Next_point): перегон между соседними пунктами.
  d(t, b, 150).
  d(t, kl, 50).
  d(t, ov, 200).
  d(ov, rz, 50).
  d(m, ka, 360).
  d(m, kl, 100).
  d(kl, dm, 120).
  d(dm, ka, 270).
```

```
  d(ka, m, 360).
  d(b, ps, 400).
  d(b, sp, 300).
  d(b, ja, 300).
  d(ja, m, 450).
  d(sp, ps, 500).
  d(sp, tn, 300).
  d(tn, ps, 400).
nx(_x,_y,_r) :-
     d( _x, _y, _r).
nx(_x,_y,_r) :-
     d( _y, _x, _r).

%% Абсолютный минимум
% Front - множество путей из _f одинаковой длины
m([ ], _) :-
   !.
m(_front,_t) :-
   new_front(_front,_nfront,_t),
   m(_nfront,_t).

new_front(_front,_nfront,_t) :-
   nfr(_front,[ ],_nfront,_t).

nfr([ ],_nfront,_nfront,_) :- !.
nfr([[_s1,[_v|_l]]|_T],_A,_nfront,_t) :-
   findall( [_s, [_nv, _v|_l]],
          (nx( _v, _nv, _d),
           not memb_check( _nv, [_v|_l] ), % без повторений; memb_check/2 -
           _s is _d + _s1,                 % функция "элемент списка"
           ifthen( _nv = _t,               % см. выше
                 (retract(found(_f,_t, _Tl)),
                  assert((found(_f,_t,[[_s, [_nv, _v|_l]]|_Tl]))))),
           _list),
   addl(_list,_A,_NA),         % добавление без повторений
nfr(_T,_NA,_nfront,_t).
%% Очередной самый короткий путь
fn(_front, _f, _t) :-
  case([_front==[ ]       % АБСОЛЮТНЫЙ МИНИМУМ найден
       -> true,
       retract(new)     % существует новый путь
       ->
          (cls,
           report(_f,_t,_),
           nl,
```

```
            write($FIND A NEW PATH? (y/n)> $),
            keyb(`n,_))                % "нет", конец
          | fail]                      % "да", перейти к новому поиску
       ).
fn( _front, _f, _t ) :-
   fnew_front(_front,_nfront,_t),
   fn( _nfront, _f, _t).

fnew_front(_front,_nfront,_t) :-
   fnfr(_front,[ ],_nfront,_t).

fnfr([ ],_nfront,_nfront,_) :- !.
fnfr([[_s1,[_v|_l]]|_T],_A,_nfront,_t) :-
   findall( [_s, [_nv, _v|_l]],
          (ctr_is(0,_C),             % счетчик 0 содержит текущую
           nx( _v, _nv, _d),         % минимальную длину пути
           not memb_check( _nv, [_v|_l] ),
           _s is _d + _s1,
           ifthen(_C > 0,
                  _s =< _C), % НАХОДИТ ТОЛЬКО ПУТИ, БОЛЕЕ КОРОТКИЕ ЧЕМ
           ifthen( _nv = _t, % МИНИМАЛЬНЫЙ в текущий момент
                 (retract(found(_f,_t, _Tl)),
                  assert((found(_f,_t,[[_s, [_nv, _v|_l]]|_Tl]))),
                  ctr_set(0,_s),
                  ifthen(not clause(new,_),
                        assert(new)))),  % найден новый путь
           _list),
   addl(_list,_A,_NA),
fnfr(_T,_NA,_nfront,_t).

%%  Вспомогательные предикаты
% добавление элемента к списку без повторений
addl([ ],L,L) :- !.
addl([E|T],A,L) :-
  not memb_check(E,A),
  !,
  addl(T,[E|A],L).
addl([_|T],A,L) :-
  addl(T,A,L).

% Вывод сообщения
report(_f,_t,_L) :-
  found(_,_,_T),
  ifthen( _T \= [ ],
       (mc(_T,_L),
```

```
        nl,
        write($The length of the shortest path from $),
        write(_f),
        write($ into $),
        write(_t),
        write($ is $),
        write(_L),
        write($ km.$))),
  nl,
  wr_rev(_T).

% вывод элементов списка в обратном порядке
wr_rev([ ]) :- !.
wr_rev([[E,L]|R]) :-
   reverse(L,L0),
   write([E,L0]),
   nl,
   wr_rev(R).

% Минимальная первая компонента списка пар:
mc([ ],0) :- !.
mc([[_f,_]|_l],_n) :-
  mc(_l,_f,_n).
mc([ ],_m, _m) :-
  !.
mc( [[_n,_]|_l],_a,_m) :-
  _n =< _a,
  !,
  mc(_l,_n,_m).
mc( [_|_l],_a,_m) :-
  mc(_l,_a,_m).
```

4.4 Логические головоломки

Особенностью логических головоломок является очень большое число вариантов условий при свободном их комбинировании, и следовательно, слишком большое пространство поиска, для того чтобы решить задачу простым перебором вариантов. Поэтому программы на Прологе, решающие логические головоломки, интересны тем, что в них используются специфические приемы быстрого отсечения бесполезных вариантов и структуры для быстрого целенаправленного поиска решения. Мы приведем решения двух головоломок, иллюстрирующие две разные техники: поиск плана действий для достижения цели, и поиск единственного решения, описываемого косвенными условиями.

Безопасная экономия перчаток

Задача: Химик работает с N активными материалами, реагирующими друг с другом, и опасными для человека. Даже то небольшое количество материала, что остается на защитных перчатках, в состоянии испортить другой материал и повредить кожу человека. Поэтому, чтобы исключить контакт с материалами и обеспечить их сохранность, по инструкции следует использовать для каждого материала новую пару защитных резиновых перчаток. И все же при дефиците перчаток желательно придумать способ, как соблюсти собственную безопасность и сохранность материалов, и при этом обойтись $M < N$ парами перчаток? Например, как, работая с тремя материалами, использовать две пары перчаток, с шестью материалами - четыре пары, и т.д.? Более точно задача ставится так: можно ли, безопасно работая с N материалами, обойтись M парами перчаток, и если можно, то как? Описать план действий как последовательность операций типа: "надеть i-ую пару", "выбросить i-ую пару", "обработать j-тый материал", и т.д. Решение головоломки основано на том, что можно использовать вывернутые наизнанку перчатки и надевать одни перчатки на другие. При этом, естественно, не всякие комбинации безопасны.

```
% safe(+Materials,+Gloves): Materials - данное число материалов, Gloves -
% данное число пар перчаток.

safe(Materials,Gloves) :-
   create(H,'plans.ari'),   % список операций будет сохранен в файл plans.ari
   forall(streat(Materials,Gloves,Plan), % каждое найденное решение
          (fwlist(H,Plan),               % вывести поэлементно в файл
           nl(H),
           nl(H))),
   close(H).

% предикат планирования
% streat(+Materials,+Gloves,-Plan): Plan - искомый список операций.
% Мы будем описывать СОСТОЯНИЕ каждой пары перчаток термом вида E:I:N,
% в котором E и I характеризуют состояние соответственно внешней и
% внутренней поверхности, и принимают одно из значений: h (контакт с
% рукой), m (контакт с материалом) и c (сторона не использована), a N -
% номер пары перчаток.

streat(Materials,Gloves,Plan) :-
  strt(Materials,0,Gloves,0,[ ],[ ],Plan).

% Рекурсия по числу материалов
% strt(+Materials,+CurrMaterial,+Gloves,+UsedNumber,+PutOn,+UsedSet,-Plan):
% Materials - количество необработанных материалов, CurrMaterial - номер
% обрабатываемого материала, Gloves - доступное количество пар перчаток,
% UsedNumber - количество использованных пар, PutOn - список надетых пар,
```

```
% UsedSet - список ненадетых использованных перчаток, Plan - искомый план.

strt(0,_,_,_,_,_,[ ]) :-
%  все материалы обработаны; конец рекурсии
  !.

strt(Materials,M,Gn,L,On,[m:m:N|Used],[throw_out(N)|P]) :-
% выбросить использованную пару, касавшуюся материалов обеими поверхностями
  !,
  strt(Materials,M,Gn,L,On,Used,P).

% Точка выбора:
strt(Ms,Cm,Gn,L,[c:D:N|On],Used,[treat(Cm1)|P]) :-
% Если наружной является пара с чистой внешней поверхностью, то МОЖНО
% обработать очередной материал
  Ms > 0,
  dec(Ms,Ms1),
  inc(Cm,Cm1),
  strt(Ms1,Cm1,Gn,L,[m:D:N|On],Used,P).  % меняется состояние поверхности

% Точка выбора:
strt(Materials,M,Gn,L,[D1:D2:N|Rest],Used,[take_off(N)|P]) :-
  D1 \== c,
% Если наружной является пара с соприкасавшейся внешней поверхностью, то
% МОЖНО снять эту пару
  strt(Materials,M,Gn,L,Rest,[D1:D2:N|Used],P).

strt(Materials,M,Gn,L,Before,[c:D:N|Used],[take_on_used(N)|P]) :-
% Источник экономии: если имеется использованная пара с чистой внешней
% поверхностью,то надеть эту пару
  !,
  take_on(Before,c:D:N,After),         % меняется состояние поверхности
  strt(Materials,M,Gn,L,After,Used,P).

strt(Materials,M,Gn,L,On,Used,[reverse(N)|P]) :-
  choose(D:c:N,Used,Rest),     % детерминизированный select/3, см. выше
% Если среди использованных пар имеется пара с чистой внутренней
% поверхностью, то вывернуть ее наизнанку
  !,
  strt(Materials,M,Gn,L,On,[c:D:N|Rest],P).

strt(Materials,M,Gn,L,Before,Used,[take_on_new(N)|P]) :-
% Если нет других возможностей, и еще имеется неиспользованная пара,
% то надеть эту пару
  N is L+1,
```

```
  take_on(Before,c:c:N,After),        % меняется состояние поверхности
  Gn > 0,
  dec(Gn,G),
  strt(Materials,M,G,N,After,Used,P).

% take_on(+Before,+NewPair,After): Before - список состояний надетых
% друг на друга перчаток (наружная - первый элемент), NewPair - новая
% пара, After - список состояний перчаток после того как надета новая
% пара; take_on описывает изменение состояний перчаток.
take_on([ ],E:D:N,[E:h:N]) :- % перчатка надевается на голую руку
  (D==c;D==h),                % внутренняя поверхность была чиста
                              % или соприкасалась с рукой
  !.
take_on([D2:X:N|R],O:D1:N1,[O:D:N1,D:X:N|R]) :-
  contact(D1,D2,D).           % изменение состояния наружной поверхности
                              % верхних перчаток и внутренней поверхности
                              % надеваемой пары
% Таблица изменения состояний:
% contact(+Inner,+Surface,-Result): Inner - внутренняя поверхность
% надеваемой пары, Surface - внешняя поверхность наружной пары до
% надевания, и Result - после надевания.
contact(_,m,m) :-
  !.
contact(m,_,m) :-
  !.
contact(h,c,h) :-
  !.
contact(c,h,h) :-
  !.
contact(D,D,D) :-
  (D==c,!;D==h).

% fwlist(+Handle,+List): поэлементный вывод элементов списка List в
% файл с указателем H
fwlist(_,[ ]) :- !.
fwlist(H,[E|R]) :-
 write(H,E),
 nl(H),
 fwlist(H,R).
```

Например, для $N = 3$ и $M = 2$ эта программа предлагает следующие два плана:

```
take_on_new(1)
take_on_new(2)
treat(1)
take_off(2)
```

```
treat(2)
reverse(2)
take_on_used(2)
treat(3)

take_on_new(1)
take_on_new(2)
treat(1)
take_off(2)
reverse(2)
treat(2)
take_on_used(2)
treat(3)
```

Следующая головоломка, известная под названием Zebra, используется как стандартный тест для оценки эффективности интерпретаторов и компиляторов Пролога, т.к. пространство поиска ее решения имеет астрономический размер.

Угадай животное

Задача: В пяти домах, раскрашенных в разные цвета, живут люди разных национальностей, курящие разные сигареты, пьющие разные напитки, и держащие в доме разных животных.

1. Англичанин живет в красном доме.
2. У испанца в доме собака.
3. В зеленом доме пьют кофе.
4. Украинец пьет чай.
5. Зеленый дом находится непосредственно справа от дома цвета слоновой кости (ivory).
6. Сигареты Уинстон (Winston) курит тот, кто держит в доме улиток (snails).
7. Сигареты Кулз (Kools) курит тот, кто живет в желтом доме.
8. Молоко пьет тот, живет в доме посередине.
9. Норвежец живет в первом доме слева.
10. Сигареты Честерфилдз (Chesterfields) курит тот, кто живет в доме, следующем за домом, где содержат лису (fox).
11. Сигареты Кулз курит тот, кто живет в доме, следующем за домом, где содержат лошадь.
12. Сигареты Лаки страйк (Lucky Strike) курит тот, кто пьет апельсиновый сок (orange juice).
13. Японец курит сигареты Парламентз (Parliaments).
14. Дом норвежца - следующий за голубым домом.

ВОПРОС 1: Кто держит в доме зебру?
ВОПРОС 2: Кто пьет воду?

Первой приходит в голову мысль решать эту головоломку с помощью стандартной техники перечисление-фильтрация, как в задаче о раскраске графа. Генератор будет перечислять все варианты значений признаков для каждого из пяти домов, а фильтр будет проверять перечисленные 14 условий. Вот как могла бы выглядеть соответствующая программа:

```
puzzle :-
    time(X),
% встроенный предикат, при вызове которого определяется текущий
% момент времени: при вызове time(T)  T унифицируется с фактом
% time(Hour,Min,Sec,Milisec)
 People=[[n(1,N1),cl(1,Cl1),c(1,C1),d(1,D1),p(1,P1)],
         [n(2,N2),cl(2,Cl2),c(2,C2),d(2,D2),p(2,P2)],
         [n(3,N3),cl(3,Cl3),c(3,C3),d(3,D3),p(3,P3)],
         [n(4,N4),cl(4,Cl4),c(4,C4),d(4,D4),p(4,P4)],
         [n(5,N5),cl(5,Cl5),c(5,C5),d(5,D5),p(5,P5)]],
% n(+Number,-Nationality), cl(+Number,-Color), c(+Number,-Cigarettes),
% d(+Number,-Drinks), p(+Number,-Pets)
     nationalities(NL),        % список национальностей
     select(N1,NL,NL1),        % угадываем национальность обитателя дома 1
     select(N2,NL1,NL2),       % угадываем национальность обитателя дома 2,
                               % исходя из остатка
     select(N3,NL2,NL3),
     select(N4,NL3,[N5]),      % остался единственный вариант
     colors(ClL),              % аналогично для цвета, и т.д.
     select(Cl1,ClL,ClL1),
     select(Cl2,ClL1,ClL2),
     select(Cl3,ClL2,ClL3),
     select(Cl4,ClL3,[Cl5]),
     cigars(CL),
     select(C1,CL,CL1),
     select(C2,CL1,CL2),
     select(C3,CL2,CL3),
     select(C4,CL3,[C5]),
     drinks(DL),
     select(D1,DL,DL1),
     select(D2,DL1,DL2),
     select(D3,DL2,DL3),
     select(D4,DL3,[D5]),
     pets(PL),
     select(P1,PL,PL1),
     select(P2,PL1,PL2),
     select(P3,PL2,PL3),
     select(P4,PL3,[P5]),
    constraints(People),    % фильтр
    time(Y),
    tdif(X,Y,Z),            % определяем длительность вычисления (см. ниже)
    create(H,'solution.ari'),
    tell_about(H,People),   % вывод в файл решения (см. ниже)
```

```
    close(H),
    nl,
    write($Runtime is $),
    write(Z),
    write($ seconds.$).

nationalities([ukrainian,englishman,spaniard,japanese,norvegian]).
cigars([winston,chesterfields,parliaments,lucky_strike,kools]).
colors([red,blue,yellow,green,ivory]).
pets([horse,snails,fox,dog,zebra]).
drinks([milk,tea,coffee,orange_juice,water]).

% предикат-фильтр
constraints(People) :-
      member([n(I,N),cl(I,Cl),c(I,C),d(I,D),p(I,P)],People),
      case([N=englishman->Cl=red]),                        % условие 1
      case([N=norvegian->I=1]),                            % условие 2
      case([N=spaniard->P=dog]),                           % условие 3
      case([N=ukrainian->D=tea]),                          % условие 4
      case([N=japanese->C=parliaments]),                   % условие 5
      case([C=winston->P=snails]),                         % условие 6
      case([C=kools->Cl=yellow]),                          % условие 7
      case([C=lucky_strike->D=orange_juice]),              % условие 8
      case([I=3->D=milk]),                                 % условие 9
      case([Cl=green->D=coffee]),                          % условие 10
      case([N=norvegian->                                  % условие 11
                       (next(I,J),
                        memchk([_,cl(J,blue),_,_,_],People))]),
      case([Cl=green->                                     % условие 12
                     (I>0,
                      J1 is I-1,
                      memchk([_,cl(J1,ivory),_,_,_],People))]),
      case([C=chesterfields->                              % условие 13
                     (dec(I,J2),
                      memchk([_,_,_,_,p(J2,fox)],People))]),
      case([C=kools->                                      % условие 14
                    (next(I,J3),
                     memchk([_,_,_,_,p(J3,horse)],People))]).

%% вспомогательные предикаты:
% Определение времени в секундах, затрачиваемого на вызов предиката. \\
% Используя встроенный предикат $time/1,$ можно определить моменты времени
% до и после вызова предиката. Остается вычислить их разность, что делает
% следующий предикат:
% tdif(+Moment_before,+Moment_after,-Difference_in_seconds)
```

```
tdif(X, Y, Z) :-
abstime(X, Xa),
abstime(Y, Ya),
Z is Ya - Xa.
% Перевод в секунды:
abstime(time(Hr,Min,Sec,Hun), Abs) :-
Abs is Hr*3600 + Min*60 + Sec + Hun/float(100).

% соседний номер
% next(+Number,-NextNum)
next(I,J) :-
    (J is I+1;
     I>0, J is I-1).

% tell_about(+H,+People): H - указатель файла
tell_about(_,[ ]) :- !.
tell_about(H,[L|People]) :-
   ptell(H,L),
   tell_about(H,People).

ptell(H,[n(I,N),cl(I,Cl),c(I,C),d(I,D),p(I,P)]) :-
    write(H,N),
    write(H,' lives in '),
    write(H,I),
    write(H,' house, whose color is '),nl(H),tab(H,5),
    write(H,Cl),
    write(H,', smokes '),
    write(H,C),
    write(H,', drinks '),
    write(H,D),
    write(H,' and pets '),
    write(H,P),
    nl(H).
```

Эта программа выполняется довольно долго. Чтобы перебирать варианты более целенаправленно, требуется более эффективная техника, требующая более пристального анализа условия задачи. Мы назвали бы эту технику *перечисление через фильтрацию*. Идея состоит в том, что вызовы вида $member(List, Multilist)$ и $select(List, Multilist, Multirest)$, где $List$ - некоторый список, в котором часть элементов - переменные, а $Multilist$ и $Multirest$ - списки списков этого вида, могут связывать часть переменных и тем самым уменьшать неопределенность. Это равносильно добавлению к системе тождеств, соответствующих мультиспису, дополнительных тождеств (с общими переменными), соответствующих списку.

```
% Быстрое решение головоломки.

fast :-
```

```
        filter(People),
        create(H,'solution.ari'),
        tell_about(H,People),
        close(H).

filter(People) :-
        time(T1),
        People=
       [[n(1,N1),cl(1,Cl1),c(1,C1),d(1,D1),p(1,P1)],
        [n(2,N2),cl(2,Cl2),c(2,C2),d(2,D2),p(2,P2)],
        [n(3,N3),cl(3,Cl3),c(3,C3),d(3,D3),p(3,P3)],
        [n(4,N4),cl(4,Cl4),c(4,C4),d(4,D4),p(4,P4)],
        [n(5,N5),cl(5,Cl5),c(5,C5),d(5,D5),p(5,P5)]],
% сначала учитывем абсолютные факты:
        [n(1,norvegian),_,_,_,_]=L1,   % условие 9
        select(L1,People,REST1),       % прочие национальности: REST1
        [_,_,_,d(3,milk),_]=L2,        % условие 8
        member(L2,REST1),!,            % фильтруем условие 8 через условие 9
        next(1,H),
        [_,cl(H,blue),_,_,_]=L3,       % условие 14
        [! member(L3,REST1) !],        % фильтруем его через предыдущие
%  [!p(X)!] равносильно q(X), где  q(X) :- p(X),!. (q - новый предикат).
        [n(H1,englishman),cl(H1,red),_,_,_]=L4,  % условие 1
        select(L4,REST1,REST2),        % фильтруем условие 1
        [n(H2,spaniard),_,_,_,p(H2,dog)]=L5,
        select(L5,REST2,REST3),     % условие 2, фильтрованное через предыдущие
        [n(H3,ukrainian),_,_,d(H3,tea),_]=L6, % условие 4
        select(L6,REST3,[L7]),         % L7 последняя оставшаяся национальность
        L7=[n(H4,japanese),_,c(H4,parliaments),_,_], % фильтрованное условие 14
        select(L7,People,REST4),       % все кроме японца
        [_,_,c(H5,winston),_,p(H5,snails)]=L8,
        select(L8,REST4,REST5),        % условие 6, фильтрованное через 14
        [_,cl(H6,yellow),c(H6,kools),_,_]=L9,
        select(L9,REST5,REST6),        % условие 7, фильтрованное через 6
        [_,_,c(H7,lucky_strike),d(H7,orange_juice),_]=L10,
        select(L10,REST6,[L11]),       % условие 12, фильтрованное через 7
                                       % L11 последняя марка сигарет
        L11=[_,_,c(H8,chesterfields),_,_],
        next(H8,H9),
       [! select([_,_,_,_,p(H9,fox)],People,REST7) !], % фильтрованное условие 10
        next(H6,H10),
        L12=[_,_,_,_,p(H10,horse)],
       [! member(L12,REST7) !],       % фильтрованное условие 11
        select([_,cl(H11,green),_,d(H11,coffee),_],People,REST8),
        H11 > 0,                       % условие 3, фильтрованное через 1
```

```
 H12 is H11-1,
 L13=[_,cl(H12,ivory),_,_,_],
[! member(L13,REST8) !],       % фильтрованное условие 5
 member([_,_,_,d(_,water),_],People),    % фильтрация вопроса 2
 member([_,_,_,_,p(_,zebra)],People),    % фильтрация вопроса 1
time(T2),
tdif(T1,T2,Sec),
nl,
write($Runtime is $),
write(Sec),
write($ seconds$),
nl.
```

Вторая программа на два порядка быстрее первой. Обе программы выводят в файл *solution.ari* следующее решение:

```
norvegian lives in 1 house, whose color is
     yellow, smokes kools, drinks water and pets fox
ukrainian lives in 2 house, whose color is
     blue, smokes chesterfields, drinks tea and pets horse
englishman lives in 3 house, whose color is
     red, smokes winston, drinks milk and pets snails
spaniard lives in 4 house, whose color is
     ivory, smokes lucky_strike, drinks orange_juice and pets dog
japanese lives in 5 house, whose color is
     green, smokes parliaments, drinks coffee and pets zebra
```

5 Пример профессиональной программы

В этом разделе мы приведем пример профессионально написанной программы на Прологе, реализующей изложенный в [9] алгоритм выделения изменяемой части русской основы. Эта программа послужит лакмусовой бумажкой: тот кто сумеет в ней разобраться, по-настоящему понял наше описание Пролога, и безусловно сумеет самостоятельно писать сложные программы на этом языке.

```
%%%%%%%%%%%%%%%%%%%% СПЕЦИФИКАЦИЯ %%%%%%%%%%%%%%%%%%%%%%%
%%%%%
%%%%%  Переключение между спецификацией и реализацией
/*
%%%%%%%%%%%%%%%%%%%%%%% ОПИСАНИЕ ЗАДАЧИ %%%%%%%%%%%%%%%%%%%%%%%%%
%% Дана парадигма, т.е. непустая последовательность словоформ.  %%
%% Требуется построить последовательность изменяемых частей     %%
%% словоформ. Входная последовательность словоформ представлена %%
%% мультисписками, т.е. списком списков букв. Решение должно    %%
%% быть представлено как список мультисписков: каждый мульти-   %%
%% список представляет разбиение изменяемой части на непрерывные%%
```

```
%% интервалы. На уровне интерфейса пользователя эти интервалы   %%
%% должны отделяться друг от друга символом "_". Например, для  %%
%% парадигмы слова "vedro", задаваемой мультисписком:           %%
%% [[v,e,d,r,o],[v,e,d,r,a],...,[v,"e,d,e,r],...,[v,"e,d,r,a,x]]%%
%% результатом будет список мультисписков:                      %%
%% [[[e],[],[o]],[[e],[],[a]],[[[e],[],[o]],[[e],[],[a]],...    %%
%% ...,[["e],[e],[]],...,[[["e],[],[a,x]]], или на уровне       %%
%% пользователя:                                                %%
%%                      e-0-o                                   %%
%%                      e-0-a                                   %%
%%                      .....                                   %%
%%                     "e-e-0                                   %%
%%                     "e-0-ax                                  %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Идея, лежащая в основе алгоритма решения этой задачи, такова. При первом проходе (т.е. параллельном просмотре списков в мультисписке) составляется список *Common* общих вхождений (в естественном порядке) букв в основы. При втором проходе очередной элемент E списка *Common* служит для нахождения границы очередной изменяемой части: она начинается там, где хотя бы в одном из списков обнаруживается элемент, не совпадающий с E. Так возникает мультисписок компонент: отрезки (возможно пустые) от начала основ до первой границы, отрезки между смежными границами, и отрезки от последней границы до конца основ.

```
%% СПЕЦИФИКАЦИЯ АЛГОРИТМА
% АЛГОРИТМ variable ПРИМЕНЯЕТСЯ К МУЛЬТИСПИСКАМ БУКВ,
variable(Mlist) :-
% ИМЕЮЩИМ ПО МЕНЬШЕЙ МЕРЕ ДВА ЭЛЕМЕНТА
  ifthen(Mlist=[_,_|_],              % парадигма не вырождена
%   1. ОТСЕКАЕТСЯ МАКСИМАЛЬНЫЙ ОБЩИЙ МУЛЬТИПРЕФИКС
    (max_multipref_sect(Mlist,Msuff),
     Msuff=[First|_],                % первый член парадигмы
%   2. ЗАТЕМ ОТ РЕЗУЛЬТАТА ОТРЕЗАЕТСЯ МАКСИМАЛЬНЫЙ ОБЩИЙ МУЛЬТИСУФФИКС
     max_msuff_cut(First,Msuff,Mrest),
%   3. ИЗ ОСТАТКА ВЫДЕЛЯЕТСЯ СПИСОК Commons ОБЩИХ БУКВ В ИХ ЕСТЕСТВЕННОМ
%      ПОРЯДКЕ
     find_commons(Mrest,Commons), % дополнительный проход для выделения
                                  % ИЗ Mlist СПИСКА Commons
%   4. ВЫДЕЛЯЕТСЯ СПИСОК PVar ИЗМЕНЯЕМЫХ ЧАСТЕЙ
     part_of_variation(Commons,Mrest,PVar),
%   5. И ПЕЧАТАЕТСЯ В ФОРМЕ ИНТЕРФЕЙСА ПОЛЬЗОВАТЕЛЯ
     surf(PVar))).                          % печать

%% выделение изменяемой части
% part_of_variation(Commons,ListOfLists,ListOfVariableParts)
part_of_variation(Comms,MList,PVar) :-
```

```
  make_multiunit(MList,MU),          % MU=[[ ],...[ ]]
% make_multiunit(List,UnitListOfLengthOfTheList): [[ ],...,[ ]]
  part_of_var(Comms,MList,MU,PVar).      % Рекурсия по Comms

% АЛГОРИТМ part_of_var ПРИМЕНЯЕТСЯ К СПИСКУ ОБЩИХ БУКВ Commons
% И К МУЛЬТИСПИСКУ Mlist, ОСТАВШЕМУСЯ ПОСЛЕ ОТСЕЧЕНИЯ МУЛЬТИ-
% ПРЕФИКСА И МУЛЬТИСУФФИКСА. ОН СТРОИТ СПИСОК ИЗМЕНЯЕМЫХ ЧАСТЕЙ
% Mpref РЕКУРСИЕЙ ПО Commons, ПОСЛЕДОВАТЕЛЬНО ОТРЕЗАЯ МАКСИМАЛЬНЫЕ
% НЕСОВПАДАЮЩИЕ МУЛЬТИПРЕФИКСЫ ОТ ОСТАТКА Mlist, И ДОБАВЛЯЯ ИХ В
% КОНЕЦ Mpref. Mpref = Result, ЕСЛИ СПИСОК Commons ПУСТ (ВЫХОД
% ИЗ РЕКУРСИИ)
% part_of_var(Commons,Mlist,Mpref,Result).
part_of_var([ ],Mlist,Mpref,Res) :-
%  1. БАЗИС: Commons=[ ] (ОПУСТЕЛ)
  ifthenelse(multiempty(Mlist),      % Mlist=[[ ],...,[ ]]
%       1.1 КОГДА ОСТАТОК МУЛЬТИСПМСКА Mlist СТАНОВИТСЯ ПУСТЫМ,
%          Mpref ЯВЛЯЕТСЯ РЕЗУЛЬТАТОМ.
       Res=Mpref,
%       1.2. КОГДА ОН НЕПУСТ, ОН ЯВЛЯЕТСЯ ПОСЛЕДНЕЙ МУЛЬТИКОМПОНЕНТОЙ
%            РЕЗУЛЬТАТА,
       (multiwrap(Mlist,Munit),
            % for the rest: [u1,...,uk] => [[u1,...,uk]]
            % multiwrap(ListOfLists,ListOf_[List]).
%           И ОН ДОБАВЛЯЕТСЯ В КОНЕЦ  Mpref ЧТОБЫ ПОЛУЧИТЬ
%           РЕЗУЛЬТАТ.
        multiconcatenate(Mpref,Munit,Res))).

%  2. ШАГ РЕКУРСИИ: СУЩЕСТВУЕТ ОЧЕРЕДНАЯ ОБЩАЯ БУКВА  C
part_of_var([C|Comms],Mlist,Mpref,Res) :-
%    2.1  C ЯВЛЯЕТСЯ ПЕРВОЙ ВО ВСЕХ СПИСКАХ ИЗ Mlist; ТОГДА ОНА УДАЛЯЕТСЯ
%         ИЗ ВСЕХ ЭТИХ СПИСКОВ БЕЗ ИЗМЕНЕНИЯ Mpref
  multifirst(C,Mlist,Mtail), % удаляет первый общий элемент в
                             % списках из ListOfLists
  !,
%         ЗАТЕМ part_of_var РЕКУРСИВНО ПРИМЕНЯЕТСЯ К Comms, ХВОСТУ
%         СПИСКА Commons, К Mtail, К ОСТАТКУ Mlist И К ТОМУ ЖЕ
%         САМОМУ Mpref.
  part_of_var(Comms,Mtail,Mpref,Res).

%    2.2  В НЕКОТОРОМ СПИСКЕ, ВХОДЯЩЕМ В Mlist, ИМЕЕТСЯ БУКВА,
%         НЕ СОВПАДАЮЩАЯ С БУКВОЙ C; ТОГДА МАКСИМАЛЬНЫЙ МУЛЬТИ-
%         ПРЕФИКС  Mlist, ПРЕДШЕСТВУЮЩИЙ C, ЯВЛЯЕТСЯ СЛЕДУЮЩЕЙ
%         МУЛЬТИКОМПОНЕНТОЙ Mpr СПИСКА Mlist
part_of_var([C|Comms],Mlist,Mpref,Res) :-
%       A. ТАКИМ ОБРАЗОМ,  Mlist РАССЛАИВАЕТСЯ НА Mpr И НА МУЛЬТИ-
```

```
%          СУФФИКС Msf, СЛЕДУЮЩИЙ ЗА  C
  multicut_pref_suff(C,Mlist,Mpr,Msf),
%       B. ЗАТЕМ МУЛЬТИКОМПОНЕНТА Mpr ДОБАВЛЯЕТСЯ В КОНЕЦ
%          Mpref, ЧТОБЫ ПОЛУЧИТЬ НОВЫЙ МУЛЬТИПРЕФИКС Mprnew
  multiconcatenate(Mpref,Mpr,Mprnew),
%       C. И  part_of_var РЕКУРСИВНО ПРИМЕНЯЕТСЯ К ХВОСТУ
%          Comms СПИСКА Commons, К  Msf, И К  Mprnew.
  part_of_var(Comms,Msf,Mprnew,Res).
% КОНЕЦ.

%% Выделение максимального мультипрефикса
% max_multipref_sect(+Mlist,-Mrest): Выделение максимального
%                                    мультипрефикса в Mlist
max_multipref_sect([[E|Tail]|MTail],Result) :-
  multifirst(E,MTail,MNTail),
  !,
  max_multipref_sect([Tail|MNTail],Result).
max_multipref_sect(Result,Result).

%% Выделение максимального мультисуффикса
% max_msuff_cut(+Pattern,+Mlist,-Mpref): выделение в квадратичное
% время максимального общего суффикса
max_msuff_cut(Pattern,Mlist,Mpref) :-
  append(_,Suff,Pattern),          % угадываем Suffix в Pattern
  mmss(Suff,Mlist,Mpref),          % проверяем, что он общий
  !.
mmss(Pattsf,[List|Mtail],[Pref|Mrest]) :-
  append(Pref,Pattsf,List),
  !,
  mmss(Pattsf,Mtail,Mrest).
mmss(_,[ ],[ ]).

%% неизменяемая часть
% Алгоритм, находящий в реальное время максимальную последова-
% тельность общих (неизменных) элементов в N входных списках
% find_commons(+MultiList,-CommonElemList).
find_commons([_],[ ]) :- !.
find_commons([First|Rest],Com) :-
  filter_out(First,Rest,[ ],Com).

% filter_out(+FirstList,+OtherLists,+CommAccumulator,-Result).
% Перебирает элементы Commons в их естественном порядке
filter_out([ ],_,Com,Rcom) :-
  !,reverse(Com,Rcom).
filter_out([E|T],Rest,Acc,Com) :-
```

```
  iterate_section(E,Rest,NRest),
% iterate_section(+Element,+ListOfLists,
%                 -ListOfTheirSuffixesFollowingTheElement)
  !,
  filter_out(T,NRest,[E|Acc],Com).
filter_out([_|T],Rest,Acc,Com) :-
  filter_out(T,Rest,Acc,Com).

%% Библиотечные предикаты спецификации
% iterate_section(+Element,+ListOfLists,
%                  -ListOfTheirSuffixesFollowingTheElement)
iterate_section(_,[ ],[ ]) :- !.
iterate_section(E,[F|T],[S|R]) :-
  section(E,F,S),
  iterate_section(E,T,R).

% section(+Element,+List,-MaxSuffixFollowingElement).
section(E,[E|T],T) :- !.
section(E,[_|R],T) :-
  section(E,R,T).

% multiempty(+[[ ],[ ],...,[ ]])
multiempty([ ]) :- !.
multiempty([[ ]|Tail]) :-
  multiempty(Tail).

% отрезает первый общий элемент в списках из ListOfLists
multifirst(_,[ ],[ ]) :- !.
multifirst(E,[[E|Tail]|IRest],[Tail|ORest]) :-
  multifirst(E,IRest,ORest).

% Расслоение списка ListOfLists на список Prefix букв,
% предшествующих данному C и список Suffix букв, следующих за C
% multicut_pref_suff(Common,ListOfLists,PrefList,SuffList).
multicut_pref_suff(_,[ ],[ ],[ ]) :- !.
multicut_pref_suff(C,[L|Tail],[P|Tail1],[S|Tail2]) :-
  cut_pref_suff(C,L,P,S),
  multicut_pref_suff(C,Tail,Tail1,Tail2).

% Расслоение в реальное время списка List на Prefix и Suffix,
% который начинается данным элементом E
cut_pref_suff(E,List,[Pref],Suff) :-
  difference_section(E,List,P-P,Pref,Suff).
difference_section(E,[E|Sf],Pref-[ ],Pref,Sf) :- !.
difference_section(E,[A|Sf],P-X,Pref,Suff) :- % P-X старый префикс
```

```
  conc(P-X,[A|Y]-Y,NP-Z),                    % NP-Z = (P conc A)-Z
  difference_section(E,Sf,NP-Z,Pref,Suff).

% make_multiunit(List,UnitListOfLengthOfList).
make_multiunit([ ],[ ]) :- !.
make_multiunit([_|T],[[ ]|R]) :-
  make_multiunit(T,R).

% multiconcatenate(ListOfLists,ListOfAppends): покомпонентный append
multiconcatenate([ ],[ ],[ ]) :- !.
multiconcatenate([L1|T1],[L2|T2],[L|T]) :-
  concatenate(L1,L2,L),
  multiconcatenate(T1,T2,T).

% multiwrap(ListOfLists,ListOf_[List]).
multiwrap([ ],[ ]) :- !.
multiwrap([L|Tail],[[L]|Rest]) :-
  multiwrap(Tail,Rest).

% Функция конкатенации списков:
% concatenate(+Flist,+Slist,?Con): Con = Flist . Slist
concatenate([ ],L,L) :- !.
concatenate([E|L],L1,[E|L2]) :-
  concatenate(L,L1,L2).
%% КОНЕЦ СПЕЦИФИКАЦИИ
*/
%%   ПЕРЕКЛЮЧАТЕЛЬ
% /*
%% Алгоритм реального времени
variable(Mlist) :-
  ifthen(Mlist=[_,_|_],    % парадигма не вырождена
       (mmps(Mlist,Msuff),
        mmss(Msuff,Mrest),
        common(Mrest,Comms),
        prtvar(Comms,Mrest,PVar),
        surf(PVar))).

%%  Отделение максимального мультипрефикса
% mmps(+Mlist,-Mrest): Выделение максимального мультипрефикса  Mlist
mmps([[E|Tail]|MTail],Result) :-
  mfirst(E,MTail,MNTail),
  !,
  mmps([Tail|MNTail],Result).
mmps(Result,Result).
```

```
%% Отделение максимального мультисуффикса
% mmss(+Mlist,-Mrest): Выделение максимального мультисуффикса Mlist
% за три прохода:
mmss(Mlist,Mrest) :-
  mreverse(Mlist,MRlist),
  mmps(MRlist,MRest),
  mreverse(MRest,Mrest).

%% неизменяемая часть
% common(+MultiList,-CommonElemList).
common([ ],[ ]) :- !.
common([_],[ ]) :- !.
common([First|Rest],Com) :-
  fltr(First,Rest,X-X,Com).

% fltr(+FirstList,+OtherLists,+CommAccumulator,-Result).
% Просматривает Commons в естественном порядке
% Аккумулятор реализуется дифференциальным списком
fltr([ ],_,Com-[ ],Com) :- !.
fltr([E|T],Rest,Acc-X,Com) :-
  itersc(E,Rest,NRest),
  !,
  addd(Acc-X,E,NAcc-Y),  % помещает E в конец дифференциального
                         % списка Acc-X
  fltr(T,NRest,NAcc-Y,Com).
fltr([_|T],Rest,Acc-X,Com) :-
  fltr(T,Rest,Acc-X,Com).

% itersc(+Element,+ListOfLists,-ListOf_sc(List))
itersc(_,[ ],[ ]) :- !.
itersc(E,[F|T],[S|R]) :-
  sc(E,F,S),
  itersc(E,T,R).

% sc(+Element,+List,-MaxSuffixFollowingElement).
sc(E,[E|T],T) :- !.
sc(E,[_|R],T) :-
  sc(E,R,T).

%% изменяемая часть
% prtvar(+Commons,+ListOfLists,-ListOfVariableParts)
prtvar(Comms,MList,PVar) :-
  mdempty(MList,MDE),              % MDE=[X1-X1,...,Xk-Xk], k=|MList|
  prtvr(Comms,MList,MDE,PVar).     % рекурсия по Comms
```

```
% prtvr(CommonRest,MList,MPrefAccDiffList,Result).
prtvr([ ],Mlist,Mpref,Res) :-
  ifthenelse(mempty(Mlist),        % Mlist=[[ ],...,[ ]]
             mdfirst(Mpref,Res),
% Первая мультикомпонента мультидифференциального списка, т.е.
% [M1-X1,...,Mk-Xk] => [M1,...,Mk]
             madd(Mpref,Mlist,Res)). % помещение с помощью madd
                                     % в конец Mpref.
prtvr([C|Comms],Mlist,MDpref,Res) :-
  mfirst(C,Mlist,Mtail),             % C является первым во всех
                                     % остатках Mlist
  !,                                 % и он удаляется
  prtvr(Comms,Mtail,MDpref,Res).

% Начало новой изменяемой компоненты
prtvr([C|Comms],Mlist,MDpref,Res) :-
  mcps(C,Mlist,Mpr,Msf),   % Mpr: новая компонента, предшествующая
                           %  C в Mlist
% Расслоение списка ListOfLists на список Prefix букв, предшест-
% вующих данному C и список Suffix букв, следующих за C
  maddd(MDpref,Mpr,MDprnew),     % добавление Mpr в конец списка
                                 % Mpref с помощью maddd
  prtvr(Comms,Msf,MDprnew,Res).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% */
%%   ПЕРЕКЛЮЧАТЕЛЬ
%%
%% Библиотечные предикаты
% покомпонентное addd
maddd([ ],[ ],[ ]) :- !.
maddd([DList|Rest],[E|Tail],[DRes|NRest]) :-
  addd(DList,E,DRes),
  maddd(Rest,Tail,NRest).

% покомпонентное add
madd([ ],[ ],[ ]) :- !.
madd([DList|Rest],[E|Tail],[DRes|NRest]) :-
  add(DList,E,DRes),
  madd(Rest,Tail,NRest).

% Первая мультикомпонента мультидифференциального списка,
% т.е. [M1-X1,...,Mk-Xk] => [M1,...,Mk]
mdfirst([ ],[ ]) :- !.
mdfirst([L-_|Tail],[L|Rest]) :-
  mdfirst(Tail,Rest).
```

```
% Отрезает первый общий элемент в списках из ListOfLists
mfirst(_,[ ],[ ]) :- !.
mfirst(E,[[E|Tail]|IRest],[Tail|ORest]) :-
  mfirst(E,IRest,ORest).

% mdempty(+List,-[X1-X1,...,Xk-Xk]), k=|List|.
mdempty([ ],[ ]) :- !.
mdempty([_|Tail],[X-X|Rest]) :-
  mdempty(Tail,Rest).

% mempty(+[[ ],[ ],...,[ ]])
mempty([ ]) :- !.
mempty([[ ]|Tail]) :-
  mempty(Tail).

% Расслоение списка ListOfLists на список Prefix букв, предшест-
% вующих данному C и список Suffix букв, следующих за C
% mcutps(Common,ListOfLists,PrefList,SuffList).
mcps(_,[ ],[ ],[ ]) :- !.
mcps(C,[L|Tail],[P|Tail1],[S|Tail2]) :-
  cps(C,L,P,S),
  mcps(C,Tail,Tail1,Tail2).

% Разделение в реальное время данного списка List на
% Prefix и Suffix, который начинается с данного элемента E
cps(E,List,Pref,Suff) :-
  dsc(E,List,P-P,Pref,Suff).
dsc(E,[E|Sf],Pref-[ ],Pref,Sf) :- !.
dsc(E,[A|Sf],P-X,Pref,Suff) :-    % P-X старый префикс
  conc(P-X,[A|Y]-Y,NP-Z),         % NP-Z = (P conc A)-Z
  dsc(E,Sf,NP-Z,Pref,Suff).

mreverse([ ],[ ]) :- !.
mreverse([List|Tail],[Rlist|Rtail]) :-
  reverse(List,Rlist),
  mreverse(Tail,Rtail).

%% Поверхностное представление
surf([ ]) :- !.
surf([List|Rest]) :-
  line(List),
  nl,
  surf(Rest).
line([ ]) :- !.
```

```
line([L]) :-
  !,
  srf(L).
line([L|Tail]) :-
  srf(L),
  write(-),
  line(Tail).

srf(L) :-
  ifthenelse(L=[ ],
     write(0),
     (list_text(L,S),
      write(S))).
```

6 Несколько слов о стилях логического программирования

Стиль программирования чаще всего определяется решаемой задачей и технологическими условиями, которые ставятся перед программистом. Автору довелось писать большие и сложные программы на Прологе, и руководить крупными проектами, выполнение которых заняло несколько лет и завершилось успешно и в срок. За эти годы в коллективах, которыми он руководил, выработались определенные требования не только к стилю, но и к оформлению программ. Мы очень кратко изложим здесь наиболее важные из них.

Главное, что технологически отличает Пролог от других языков программирования - это:

1) наличие возвратов,
2) неизбежно глубокая вложенность рекурсивных вызовов,
3) преимущественное использование локальных переменных,
4) бестиповость,
5) зависимость видов параметров процедур от контекста вызова.

Поэтому особенности стиля программирования на Прологе связаны именно с этими обстоятельствами.

Наличие возвратов может превратить в пытку отладку большой неряшливо написанной программы. Опыт показывает, что сложную программу правильно разрабатывать одновременно и сверху (от процедур к подпроцедурам), и снизу (от библиотеки подпрограмм к вызывающим процедурам). Можно добиться простоты управления возвратами, постепенно создавая библиотеку небольших вспомогательных программ, отлаженных независимо и при прямом выполнении, и при возврате, и одновременно, отлаживая (в обоих режимах) вызывающую программу, в которой вызовы отсутсвующих библиотечных предикатов заменены заглушками. Если ожидается, что библиотечный предикат $p(X)$ создает точку выбора, то и заглушка может быть определена несколькими фактами:

```
p(t1).
.....
p(tN).
```

Очень полезно наглядно "инкапсулировать" возвраты, используя конструкции типа [! ... !] (см. программу "Zebra"). В самом деле, в последовательности вызовов $p(\ ), [!\ q(\ )\ !], r(\ )$ непосредственно видно, что возврат из вызова $r(\ )$ "обходит" вызов $q(\ )$, и попадает сразу в вызов $p(\ )$. Это позволяет, например, писать прозрачные для понимания и отладки вложенные циклы-repeat:

```
proc :-
  repeat,
  [! loop0(X) !],
  until0(X).

until0(X) :-
  (condition0(X),
   !,fail;                % следующий шаг цикла
   true).                 % выход из цикла
```

Глубокая вложенность вызовов, особенно в больших программах, где она может достигать нескольких сотен и даже тысяч, предъявляет повышенные требования к виду рекурсии, требует оптимизации рекурсии. Чаще всего приходится выбирать между хвостовой рекурсией или циклами возвратами. Если организация цикла не связана с использованием глобальных переменных, то цикл может оказаться предпочтительнее. Более того, бывают задачи, в которых циклы значительно естественнее, нежели рекурсия, например, при программировании интерфейсов пользователя удобны "бесконечные" циклы-repeat, в которых тело организованно как оператор *case*, и каждая ветвь оператора отвечает обработке отдельного события во входном потоке:

```
loop(X) :-
  get0(X),
  case([
    test1(X) -> branch1(X),
     ...
    testK(X) -> branchK(X)|
    else_branch(X) ]).
```

Следует иметь в виду, что преобразование определения к хвостово-рекурсивному виду может сделать его менее прозрачным (ср. наивный синтаксический анализатор и его версию с дифференциальными списками.) Поэтому по возможности следует использовать прозрачное, хотя и неоптимальное, рекурсивное определение в качестве спецификации, а впоследствии - комментария к оптимальному окончательному определению (именно так сделано в программе выделения изменяемой части основы).

Преимущественное использование локальных переменных с одной стороны делает программы более прозрачными (процедуры практически независимы), с другой же стороны, в случаях, когда использование глобальных переменных неизбежно, а программа критична по времени исполнения, приходится оценивать способы их определения и размещения.

Дело в том, что факты или другие структуры БД Пролога, хранящие значения глобальных переменных, не компилируются. Поэтому, если такая переменная должна интенсивно использоваться, то ее следует хранить в той части БД, которая находится в оперативной памяти. Надо сказать, что при известной изобретательности можно практически обойтись без таких переменных.

Бестиповость и зависимость видов параметров процедур от контекста их вызова может приводить к очень трудно обнаруживаемым ошибкам. Как ни странным это может показаться, дисциплина минимальной спецификации предикатов, принятая в этом курсе лекций, оказывается весьма эффективным подспорьем. Между тем, промышленный стиль программирования отличает стандарт спецификаций, разделение спецификаций алгоритмов и программ, и высокая степень подробности. Спецификация пишется в расчете на читателя, не знакомого с программой, но нередко она необходима и автору. Опыт показывает, что уже через 3-4 месяца автор сложной программы понимает ее не лучше другого программиста. Поэтому нормальное соотношение размеров текста программы и ее спецификации 1 к 2 (см. приведенную выше промышленную программу).

В заключение отметим, что даже прагматический промышленный стиль программирования на Прологе отличает логичность и прозрачность, таково влияние языка. Разработка программ на Прологе в несколько раз быстрее разработки программ того же класса на Си. Легенды о непригодности Пролога для промышленных приложений не выдерживают критики. Современные методы компиляции Пролога дают код вполне сравнимый по быстродействию с кодом лучших компиляторов Си, и более оптимальный нежели, например, код компиляторов современных Visual-языков. Более того, современные системы логического программирования открыты для Си в обе стороны, что позволяет выбирать гибкий баланс между Си и Прологом в одной программе. И, наконец, последнее - для программирования задач целенаправленного перебора и поиска Пролог несомненно эффективнее любого другого языка. Даже очень сильный и опытный программист не в состоянии конкурировать со сложнейшей и многоуровневой оптимизацией вычисления и кода, заложенной в Абстрактной Машине Уоррена [7] - системе команд компилятора Пролога.

Литература

[1] Клини С.К.: Математическая логика. *М., "Мир".* (1973).

[2] Мендельсон Э.: Введение в математическую логику. *М., "Наука".* (1971).

[3] Lloyd,J.W.: Foundations of Logic Programming. *Springer-Verlag.* (1984).

[4] Чень, Ч., Ли, Р.: Математическая логика и автоматическое доказательство теорем. *М. "Наука".* (1983).

[5] Хоггер, К.: Введение в логическое программирование. *М., "Мир".* (1988).

[6] Стерлинг, Л., Шапиро, Э.: Искусство программирования на языке Пролог. *М., "Мир".* (1990).

[7] Ait-Kaci, H.: Warren's Abstract Machine. A tutorial. *The MIT Press.* (1991).

[8] И.Братко: Программирование на языке Пролог для искусственного интеллекта. *М., "Мир".* (1990).

[9] А.А.Зализняк: Русское именное словоизменение. *М., "Наука".* (1967).