Приношу извинения за (возможно необоснованное) использование шрифта Symbol.

# 4. Программы как теории.

*Что же касается алгоритмов, то надлежало лишь открыть*
*такую группу чудовищ, замкнутую счетную подгруппу*
*которой составляла бы собственно полиция.*

Станислав Лем "Кибериада"

## «Эвристическая машина».

При обучении основам программирования довольно популярна метафора Исполнителя - эдакого туповатого, но пунктуального и неутомимого создания способного выполнять некоторый набор команд, и что особенно важно поддающегося обучению. Для того чтобы обучить Исполнителя новой команде, необходимо составить инструкцию по ее исполнению (программу), то есть описать эту команду в терминах уже известных. В процессе выполнения программы Исполнитель может давать команды другим исполнителям, иногда даже имеется возможность клонирования и частичного переобучения исполнителей. По мере формализации Исполнитель несколько теряет свой колорит и превращается в абстрактную машину, а там и вовсе в набор операций на пространстве состояний.

Для описания декларативного программирования удобно использовать другую метафору - эвристическую машину (ЭвМ) для решения инженерных, научных, социологических и иных проблем. Это гениальное творение Эдельвейса Захаровича Машкина[1] содержит внутре анализатор и думатель, благодаря чему может использоваться для отвечания на любые вопросы. К сожалению, современный уровень развития науки и техники не позволяет пока производить устройства, непосредственно обращающие материю вопроса в синекдоху отвечания. Поэтому мы рассмотрим возможность создания не столь совершенной, но все ещё полезной машины, которую назовем псевдоэвристической машиной (ПЭвМ). Эта машина будет отличаться от ЭвМ рядом упрощений. Во-первых, вместо естественного человеческого языка, например русского для общения с ПЭвМ будет использоваться менее могучий формализованный язык. Это позволит резко упростить конструкцию анализатора. Во-вторых, высокоинтеллектуальный процесс думанья заменим процессом логического вывода или подобным процессом формального манипулирования символьными выражениями. Наконец не надеясь на самообучение машины, мы будем предоставлять ей всю необходимую информацию, то есть будем явно описывать все, что имеет отношение к интересующей нас проблеме.

Очевидно, что язык для общения с ПЭвМ должен содержать, по крайней мере, две категории выражений: вопросы для спрашивания и ответы для отвечания. Между этими классами существует тесная взаимосвязь. Любой вопрос подразумевает определенную форму ответа. Например, выражение "2*2" может интерпретироваться как вопрос "чему равно 2*2?". На такой вопрос естественно ожидать ответа "2*2 = 4" или просто "4". В то же время "2*2 = 5" также можно считать ответом, хотя и неправильным, а вот предложение "2*2 = 2+2", хотя и верное, ответом не является. Другой вид вопроса "2*2 = 4" подразумевает два варианта ответа "да" или "нет". Итак, с одной стороны необходимо установить, какие выражения в силу своей синтаксической структуры являются ответами (необязательно правильными) на поставленный вопрос, а с другой - определить понятие правильного ответа[2].

Имеется очевидная аналогия между ПЭвМ и традиционной абстрактной машиной. Так, вопросам соответствует ввод, ответам - вывод, причем вывод зависит как от ввода, так и от информации имеющейся у машины, то есть от программы. Разница проявляется, прежде всего, в форме программ. В ПЭвМ программа представляет собой совокупность утвердительных предложений, описывающих предметную область.

Для начала, предположим, что грамматика языка уже задает правила построения предложений. Таким образом, определено (как правило, бесконечное) множество всех предложений языка, то есть всех осмысленных высказываний. Произвольное подмножество этого множества называется *теорией*. Интуитивно: теория - множество "истинных" предложений или, другими словами, теория чего-то - всё, что мы можем сказать по данному поводу.

Самый простой способ определить теорию - просто перечислить все составляющие ее предложения. Понятно, что таким образом можно определить только совсем тривиальные теории. В математике (а в меньшей степени и в других науках) популярен аксиоматический метод построения теорий. Формулируются некоторые основные положения (аксиомы или постулаты) и все утверждения данной теории получаются "дедуктивным методом", то есть путём логического вывода.

Описанные методы по своей сути чисто синтаксические. Они определяют теорию без учёта смыслового содержания предложений. Принципиально иной способ определить теорию - семантический, основанный на связи языка с действительностью (или нашим представлением о ней). Выражениям языка приписывается определенный смысл в терминах описываемой им предметной области - *модели*. Такое приписывание называется *интерпретацией* языка. Предложениям языка ставятся в соответствие утверждения об объектах модели, которые могут быть как истинными, так и ложными. Множество всех предложений, которым соответствуют истинные высказывания, и образует теорию данной предметной области.

Аксиоматически определенные теории гораздо удобнее для формальных манипуляций, в том числе и для автоматической обработки. Поэтому для математики характерен процесс аксиоматизации - как только свойства некоторой математической структуры достаточно хорошо изучены, их пытаются выразить в небольшом наборе аксиом.

Этот процесс аксиоматизации во многом аналогичен процессу программирования. Создание программы начинается с изучения проблемы, которую требуется решить и формализации её в виде абстрактной модели. Описание модели в терминах языка программирования задает интерпретацию языка. Основываясь на этой интерпретации, создаётся набор аксиом, описывающий теорию модели, который и составляет логический компонент программы. Конечно, на практике модель и интерпретация редко выписываются в явном виде, но в принципе это всегда можно сделать.

Теперь можно расширить язык, добавив к предложениям вопросы и ответы. Обычно различают два типа вопросов: уточняющие (такие как "Есть ли жизнь на Марсе?") и восполняющие ("Что у ей внутре?"). Любому предложению P языка естественным образом ставится в соответствие уточняющий вопрос "действительно ли P?" предполагающий два варианта ответа "да" или "нет". Правильным ответом будет "да", если P принадлежит теории и "нет" в противном случае. Восполняющий вопрос можно построить, заменив в предложении некоторые члены специальными заместителями, обычно называемыми неизвестными или переменными (в данном случае первый термин более соответствует сути дела). Тогда предложению P содержащему неизвестные $x_1,...x_n$

соответствует вопрос "для каких $x_1,...x_n$ выполняется P?" Ответами могут служить все предложения, получающиеся при подстановке вместо неизвестных допустимых выражений, а правильными будут те из них, которые входят в теорию. На случай если таковых не окажется надо предусмотреть специальный вид ответа. Впрочем, зачастую удобнее в качестве ответа использовать сами подстановки. Остаются проблемы с примерами вида "2*2". Это выражение можно считать сокращенной записью вопроса "2*2 = x". Ясно, что теория целых чисел содержит бесконечное множество предложений могущих служить ответами на этот вопрос, в то время как нас интересует только один. Следовательно, необходимо как-то ограничить вид ответов, выбрать из них "самые простые". Сделать это в общем виде нелегко, но при разработке конкретных языков определяется "нормальная форма" выражений допустимых в качестве ответов.

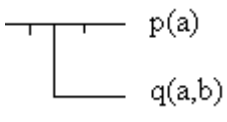
## Логические языки и их семантика.

Разнообразие решаемых задач чрезвычайно велико и создать универсальный язык, пригодный для описания любой из них практически невозможно. Конечно, можно разрабатывать специализированные языки для отдельных классов задач, и в некоторых случаях это целесообразно. Но мы пойдем на хитрость и упростим задачу, создав язык для описания математических моделей задач. Тем самым мы возложим на чужие плечи труд по составлению такой модели, воспользовавшись популярным в сфере ИТ принципом "спихнуть работу на юзера". Заодно применим и принцип "повторного использования", воспользовавшись опытом, накопленным более чем за два тысячелетия развития логики и математики. Этот опыт подсказывает, что для моделирования столь угодно сложных взаимосвязей достаточно двух понятий - функции и отношения. На самом деле достаточно и одного из них, но мы не будем слишком экономными.

Как уже говорилось, это будет *формализованный язык*, то есть язык со строго определенными правилами построения *выражений* - обычно цепочек символов из некоторого *алфавита*. Правила построения выражений определяются заданием формальной грамматики, либо другим способом, позволяющим выяснить, является ли данная цепочка "правильной" (и возможно определить её грамматическую категорию). Точным описанием языков занимается теория формальных языков. Мы же ограничимся абстрактным синтаксисом языка. То есть будем принимать во внимание только грамматические категории выражений и структуру их вложенности, отвлекаясь от лексики и правил записи выражений в виде последовательности символов. Например, три цепочки символов:

1. $p(a) \wedge q(a,b)$
2. Kpaqab
3. (and (p a) (q a b))

можно рассматривать как различные текстовые представления одного и того же "абстрактного" выражения. Представление не обязательно должно быть текстовым. Это же выражение могло бы выглядеть и так[3]:

4. 


Описывая предметную область, мы говорим, прежде всего, о некоторых *сущностях*, которые будем называть также предметами и *объектами* (невзирая на попытки узурпировать этот термин со стороны ООП). Эти сущности образуют множество,

именуемое универсумом рассмотрения. На этом множестве определяются *функции* и *предикаты* (свойства и отношения), выражающие взаимосвязи между сущностями. На всякий случай уточним, что термином функция мы обозначаем не то безобразие, которое называют этим словом в Си, а математическое понятие, выражающее идею однозначного соответствия.

Чтобы иметь возможность формировать утверждения введем в язык символы - имена для объектов, функций и предикатов. Собственные имена объектов называют предметными константами, а функций и отношений соответственно функциональными и предикатными (реляционными) константами. Каждой функциональной и реляционной константе приписано число называемое ее *арностью* (а также местностью или рангом). Часто предметные константы удобно рассматривать как нульарные функциональные. Из предметных и функциональных константа формируются сложные имена объектов - *термы*. Терм определяется как выражение вида $f(t_1,...,t_n)$, где f - n-арная функциональная константа, а $t_1,...,t_n$-термы и обозначает объект, получаемый применением функции f к объектам $t_1,...,t_n$. *Элементарное предложение* - выражение вида $r(t_1,...,t_n)$, где r - n-арная реляционная константа, а $t_1,...,t_n$-термы. Это предложение выражает мысль, что объекты $t_1,...,t_n$ находятся в отношении r, а в случае n = 1 - что объект обладает свойством r. Нульместные реляционные константы (пропозициональные константы) представляют атомарные высказывания, то есть такие высказывания, внутренняя структура которых несущественна. Более сложные предложения строятся с применением *логических связок*: отрицания ($\neg$), конъюнкции ($\wedge$), дизъюнкции ($\vee$), импликации ($\Rightarrow$), эквивалентности ($\Leftrightarrow$).

Описанный язык позволяет делать только весьма конкретные утверждения относительно отдельных объектов. Чтобы добиться большей общности введем специальные символы - *переменные*, которые можно подставлять вместо имен объектов и таким образом превращать предложения в высказывательные формы (*формулы*). Такая формула уже не выражает определенной мысли, но ее снова можно превратить в предложение, связав переменные *кванторами*. Традиционно в логике употребляется два вида кванторов: всеобщности ($\forall$) и существования ($\exists$). Если F некоторая формула, содержащая единственную переменную x, то ($\forall x$: F) - предложение утверждающее, что F выполняется для всех рассматриваемых объектов, а ($\exists x$: F) - по крайней мере, для одного из них. Таким образом, появляется возможность описывать взаимосвязи между предикатами. Например, делать такие знаменитые утверждения, как

($\forall x$ : человек(x) $\Rightarrow$ смертен(x))
или
($\forall x$: $\forall y$: управдом(x) $\wedge$ человек(y) $\Rightarrow$ друг(x,y) ).

Довольно часто квантификация переменных производится неявно. Знакомая со школы формула $(x + y)^2 = x^2 + 2xy + y^2$ утверждает истинность равенства для любых значений x и y и с точки зрения формальной логики должна записываться с использованием кванторов всеобщности. Но по общепринятым соглашениям свободные вхождения переменных трактуются как указание на всеобщность. То есть приведенная формула считается сокращением для ($\forall x$:$\forall y$: $(x + y)^2 = x^2 + 2xy + y^2$ ).

Полученный язык обычно называют *языком первого порядка*, а теории, им описываемые *теориями первого порядка*. Он позволяет описывать определенный класс моделей - алгебраические системы. Это одна из наиболее общих математических структур - значительная часть математики посвящена исследованию их разновидностей.

*Алгебраическая система* представляет собой множество с определенными на нем функциями и отношениями (предикатами). Если система содержит только отношения, то ее называют *реляционной системой* (или моделью, но этот термин и так перегружен), а если только операции - *(универсальной) алгеброй*.

Интерпретации языка определяются путём сопоставления имен с объектами системы. Предметные константы отображаются на элементы, функциональные на функции, реляционные на отношения. Таким образом, каждое предложение превращается в утверждение относительно объектов системы и имеет смысл говорить о его истинности или ложности[4]. Тот факт, что предложение P истинно в M записывают как $(M)\models P$. Все истинные предложения образуют теорию данной системы. Можно даже формально определить

$$Th(M) = \{P : (M)\models P\}.$$

Обратно, для любой теории можно определить класс всех ее моделей, то есть таких интерпретаций, при которых все предложения теории истинны

$$Mod(T) = \{M : \forall P.\ P \in T \Rightarrow (M)\models P\}$$

Это дает нам основания ввести отношение *логического следования*. Множество предложений $T_2$ (логически) следует из множества предложений $T_1$ ($T_1 \models T_2$),если любая модель $T_1$ является моделью и для $T_2$.

$$(T_1 \models T_2) \equiv Mod(T_2) \subset Mod(T_1)$$

В случае если $T_2$ состоит из одного предложения, это определение принимает вид: предложение P следует из множества предложений T ($T \models P$) если P истинно в любой модели T.

Некоторые предложения, будут истинными при любой интерпретации. Такие предложения называются *тождественно истинными*, или *тавтологиями.* Утверждение "P - тавтология", кратко записывают как $\models P$, что можно понимать как $\varnothing \models P$.

Множество всех тавтологий, образует теорию. Эта теория называется *логикой первого порядка*. Иногда ее называют и *логикой предикатов*, но, как правило, последний термин относится к теориям реляционных систем, то есть к теориям, язык которых не содержит функциональных констант. С другой стороны, можно попытаться обойтись без реляционных констант. Понятно, что совсем без них нельзя, ведь необходимо как-то формировать предложения. Но достаточно всего одной, обычно обозначаемой "=" и интерпретируемой как отношение *равенства*, то есть идентичности двух объектов. Это отношение настолько важно и полезно, что часто его включают в понятие логики первого порядка, хотя все же полезно разделять "чистую" логику первого порядка и *логику первого порядка с равенством*. Если же все предложения логики представляют собой равенства, то ее называют *эквациональной логикой*[5].

Довольно часто совокупность рассматриваемых предметов естественным образом разбивается на классы, причем различные отношения имеют смысл только для объектов определенных классов. Таким образом возникают *многосортные алгебраические системы* и соответственно *многосортные логики*. С каждой предметной константой и переменной связывается определенный *сорт* или *тип*, а с функциональной и реляционной - *сигнатура*, то есть набор типов аргументов. Синтаксически правильными считаются

только те формулы, в которых соблюдается соответствие типов. При интерпретации каждому типу ставится в соответствие определенное множество объектов - *домен* типа. Главное достоинство многосортных логик - возможность чисто синтаксически определять бессмысленность некоторых выражений. Кроме того, появляется возможность унифицировать понятия функции и отношения. Для этого достаточно выделить определенный "логический" тип, представляющий множество {ложь, истина} и идентифицировать отношения с функциями логического типа. К сожалению, вместе с заведомо бессмысленными выражениями под запретом оказываются и вполне разумные. Так, теряется возможность определять общие функции и отношения. Определенного компромисса удается добиться, усложняя систему типов, например, вводя отношение порядка между типами, которому соответствует отношение включения между доменами.

Термин "логика первого порядка" наводит на мысль, что возможна логика "не первого порядка". Действительно, и функции и отношения можно рассматривать как сущности, которые сами могут быть аргументами функции и отношений. Можно ввести предикатные или функциональные переменные и кванторы по этим переменным. Ещё одна интересная возможность - формирование абстракций по выражениям, например "объект, обладающий свойством p" ($\tau x : p(x)$), или множество таких объектов $\{x : p(x)\}$.

Такого рода теории носят общее называние "логики высшего порядка". Их многообразие чрезвычайно велико. Собственно, ещё система Фреге допускала функциональные переменные и кванторы по ним, (то есть была логикой второго порядка) что позволяло представлять утверждения невыразимые в языках первого порядка, например, дать определение равенства:

$$x = y \equiv (\forall p : p(x) \Leftrightarrow p(y)\ ).$$

Однако при построении таких систем возникают значительные трудности. Так система Фреге оказалась противоречивой, как и многие другие подобные логики. Чтобы избавится от противоречий приходиться накладывать ограничения на язык этих теорий. Наиболее плодотворным из них оказалось введение типизации, поэтому многие логики высшего порядка получили название "теории типов". Первая полностью формализованная логика высшего порядка - "простая теория типов" Черча породила множество вариаций. Одну из них мы рассмотрим[6].

Язык теории содержит две категории выражений типы и термы, причем каждому терму соответствует тип. Каждый тип A является именем некоторого множества $D_A$ - домена типа. При построении теории исходят из базового семейства типов, которые включены в алфавит языка.

Множество всех типов определяется индуктивно на основе базовых типов с использованием символов "$\rightarrow$" и "$\times$".

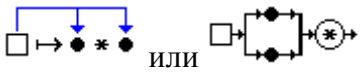
1. Если A и B - типы, то $A \times B$ - тип, представляющий декартово произведение соответствующих доменов $D_A \times D_B$.
2. Если A и B - типы, то $A \rightarrow B$ - тип. Соответствующий ему домен - множество отображений из $D_A$ в $D_B$

Кроме типов, алфавит языка содержит константы и переменные, с каждой из которых связан тип. Термы строятся по следующим правилам:

1. Константа или переменная типа A - терм типа A.

2. Если a и b - термы типа A и B соответственно, то <a,b> - терм типа A×B.
   Терм <a,<b,c>> условимся записывать как <a,b,c>. Аналогично для любых $a_1,\ldots a_n$ запись $<a_1,\ldots a_n>$ будет означать $<a_1,<a_2,\ldots a_n>...>$.
3. Если f - терм типа A→ B и a - терм типа A, то f(a) - терм типа B.
   Терм $f(<a_1,\ldots a_n>)$ будем записывать в виде $f(a_1,\ldots a_n)$. Таким образом, хотя все функции одноместны, можно использовать привычный синтаксис многоместных функций.
4. Если t - терм типа B и *x* - переменная типа A, то (λ *x*.t) - терм типа A→B.

Последнее правило позволяет вводить сложные имена функций, то есть определять функции, не давая им явных имен[7]. Говорят, что переменная x *связана*, если она встречается внутри выражения вида (λ *x*.t) и *свободна* в других выражениях. Терм, не содержащий свободных переменных, называется *замкнутым*, или *основным*. Только основным термам можно приписать определенный смысл и целью введения переменной является ее последующее связывание. Например, выражение (λ x . x*x ) описывает функцию возведения в квадрат. По сути, оно уже не содержит переменных и может быть представлено в виде:




Алфавит языка содержит предопределенный тип **B**, доменом которого является двухэлементное множество истинностных значений {ложь, истина} и стандартные логические константы:

- **T** и **F** типа **B**.
- ¬ типа **B**→ **B**.
- ∧,∨,⇒,⇔ типа (**B** × **B**) → **B**.

Кроме того, для каждого типа A, имеются следующие константы.

- $=_A$ типа (A×A) → **B** - отношение равенства на $D_A$.
- $\Sigma_A$ и $\Pi_A$ типа (A→**B**) → **B**.

Термы типа **B** называются формулами, а замкнутые формулы - предложениями. Функция f типа (A→ **B**) является предикатом на A и выражает некоторое свойство объектов типа A. Тогда Π(f), означает что f всюду принимает значение **T**, то есть что это свойство выполняется для всех объектов типа A. Σ(f) означает что f принимает значение **T**, по крайней мере, в одном случае, то есть существует хотя бы один объект, обладающий этим свойством. Термы вида Σ(λ*x*.t) и Π(λ*x*.t) можно записывать как (∃*x*: t) и (∀*x*: t) поскольку они выражают именно этот смысл. Определение модели теории и отношения логического следования выполняется так же как для логики первого порядка. Имеется, однако, одна тонкость. Часто рассматривают *нестандартные* модели, в которых A→B интерпретируется не как множество всех функций из A в B, а как множество (рекурсивно) вычислимых функций.

## Формальные системы.

Выбрав определенную логику, мы можем описать интересующую нас проблемную область набором утверждений. Отношение логического следования определяет, какие предложения будут верными следствиями этих утверждений. Однако ведь мы

предполагаем, что машина ничего не знает об интерпретациях и моделях. Здесь на помощь приходит аксиоматический метод, заменяющий содержательные рассуждения чисто формальным манипулированием выражениями. Формализация аксиоматического метода приводит к понятию *формальной системы* или *исчисления*.

Формальная система состоит из формализованного языка и механизма вывода. Последний, в свою очередь, состоит из *правил вывода*, которые позволяют из одних предложений получать другие и некоторого множества предложений, называемых *аксиомами*. Последовательность предложений, каждое из которых либо является аксиомой, либо получено из предыдущих посредством одного из правил называется *формальным выводом* или *доказательством*, а предложения, получаемые в процессе вывода - *теоремами*. Утверждение "P - теорема" записывается как |– P.

Заметим, что формальная система - довольно широкое понятие. Например, формальные грамматики представляют собой разновидность исчислений ("теоремами" грамматики будут цепочки ею порождаемые). Такое определение несколько противоречит интуитивному пониманию слова "теория". Нас будут интересовать системы построенные на основе логических языков и, прежде всего тех, которые мы рассмотрели.

Добавляя к аксиомам формальной системы новые аксиомы, мы получим новые теоремы. Таким образом устанавливается отношение выводимости. Предложение P *выводимо* из множества предложений T (T |– P), если оно становится теоремой при добавлении к множеству аксиом посылок T.

Весьма желательно чтобы исчисление позволяло выводить все истинные утверждения. Это свойство называется *полнотой* исчисления. Ещё более важна *корректность* - выводимыми должны быть только истинные утверждения. Если исчисление корректно и полно, то отношения логического следования (T |= P) и выводимости (T |– P) эквивалентны. Несколько хорошо известных аксиоматических систем для логики первого порядка обладают обоими этими свойствами. Хуже обстоит дело с логиками высших порядков. Вследствие теоремы Геделя о неполноте[8] эти логики не могут быть вполне аксиоматизированы. Но если при построении моделей ограничится только вычислимыми функциями, полное исчисление можно построить и для них.

Наличие формальных правил вывода позволяет автоматически проверять правильность доказательств, но не дает метода находить эти доказательства. Таким образом, остается неясно как искать ответ на поставленный вопрос. Идеальным решением было бы наличие *разрешающей процедуры* - алгоритма определяющего принадлежит ли предложение данной теории. К сожалению, известные разрешимые исчисления - логика высказываний и логика классов (то есть логика одноместных предикатов) оказываются очень ограниченными в выразительных возможностях, а сколько-нибудь интересные теории принципиально неразрешимы. Впрочем, этот факт не должен нас смущать - любая теория которая способная описать работу, скажем, машины Тьюринга будет неразрешимой. Лучшее, что может гарантировать любой метод, если решение существует, то оно будет найдено. Наивный подход к реализации разрешающей процедуры - исходя из аксиом и пользуясь правилами вывода получать все новые и новые следствия, пока не получим интересующее нас предложение. Неэффективность такого метода очевидна. Гораздо больше обещает обратный метод: взять интересующее нас предложение и выяснить из каких посылок оно может быть получено, затем проанализировать эти посылки и т. д., пока не придем к аксиомам. Однако, и на этом пути нас подстерегают трудности в виде так называемых "комбинационных взрывов" - неконтролируемого роста объемов вычислений.

**Ограничения реальных языков.**

При разработке реальных языков программирования приходиться идти на компромисс и жертвовать полнотой, а иногда и корректностью. (Хотя последнее крайне нежелательно и обычно нецелесообразно. Как показала история развития Лиспа, ведущее к некорректности динамическое связывание переменных оказалось и менее эффективным, хотя поначалу предполагалось обратное.) Кроме того, известные в настоящее время методы реализации требуют существенных ограничений как для формул, которые задают теорию, так и для формул использующихся в запросах.

Большинство логических языков программирования основаны на подмножествах логике первого порядка. В Прологе используется логика хорновских предложений. Эти предложения (называемые также определительными предложениями или хорновскми дизъюнктами) имеют вид:

$$G_1 \wedge ... \wedge G_n \Rightarrow H$$

или, более строго

$$(\forall x_1 ... x_m: G_1 \wedge ... \wedge G_n \Rightarrow H),$$

что эквивалентно

$$(\forall x_1 ... x_k: (\exists x_{k+1} ... x_m: G_1 \wedge ... \wedge G_n) \Rightarrow H),$$

где H и $G_1 ... G_n$ - элементарные формулы, $x_1 ... x_m$ - переменные, входящие в формулы, причем $x_1 ... x_k$ входят в H, а $x_{k+1} ... x_m$ -"локальные" переменные, входящие в $G_1 ... G_n$.

Чтобы стало понятнее, рассмотрим популярный пример из области семейных отношений. Если существует некоторый z, такой, что x - сын z , а z - сын y, то x - внук y. Формально это можно записать в виде

$$(\forall x: \forall y: (\exists z: \text{сын}(x,z) \wedge \text{сын}(z,y) \Rightarrow \text{внук}(x,y))$$

или, может быть менее естественно,

$$(\forall x: \forall y: \forall z: \text{сын}(x,z) \wedge \text{сын}(z,y) \Rightarrow \text{внук}(x,y))$$

опуская кванторы, получаем

$$\text{сын}(x,z) \wedge \text{сын}(z,y) \Rightarrow \text{внук}(x,y)$$

что на Прологе записывается как

```
внук(x,y) :- сын(x,z) , сын(z,y).
```

Набор таких предложений с одним и тем же реляционным символом в заголовке рассматривается как определение отношения, обозначаемого этим символом. В Прологе и сходных с ним языках именно отношения определяют вычисления (это и дает основания называть их реляционными языками), в то время как функциональные символы служат для описания данных. Хотя хорновские предложения довольно сильно ограничены, показано, что ними можно определить любое вычислимое отношение.

Запросы к логической программе имеют вид "$G_1 \wedge ... \wedge G_n$". Если такая формула не содержит переменных, то она является предложением и ответом может быть "да" или "нет". Если же формула содержит переменные $x_1,...x_m$, то она интерпретируется как задание найти такие значения для $x_1,...x_m$, которые превратят формулу в истинное высказывание, а ответами будут наборы этих значений (или "нет", если таких ответов не существует).

В информатике есть ещё одна область, связанная с отношениями - теория реляционных баз данных. Реляционную базу данных можно рассматривать как множество элементарных предложений логики предикатов, то есть выражений вида $P(c_1,...,c_n)$, где P - n-местный предикат, а $c_1...c_n$- константы. Каждое такое предложение утверждает, что объекты с именами $c_1...c_n$ находятся в отношении P. (Специалисты по базам данных предпочитают говорить, что кортеж $<c_1,...,c_n>$ принадлежит отношению P, но сути дела это не меняет.) Запросы к базе данных имеют вид "для каких $x_1,...,x_n$ выполняется F?", а ответами служат наборы значений для $x_1,...,x_n$, делающие F истинным высказыванием. Поскольку формулы, определяющие базу данных имеют очень простой вид, ограничивать форму запросов не требуется и, как правило, в качестве F допускаются произвольные формулы логики предикатов.

Теория реляционных баз данных разрабатывалась совершенно независимо от развития Пролога и логического программирования. Тем не менее, их близкая взаимосвязь привлекла внимание специалистов, что привело к созданию прологообразного языка запросов Datalog. В этом языке форма запросов ограничена конъюнкциями элементарных формул, но допускается определять дополнительные отношения посредством хорновских предложений, что, по мнению авторов языка, позволяет более естественно выражать смысл сложных запросов.

В основе языков функционального программирования лежат эквациональные логики. Здесь также приходиться ограничивать форму уравнений, определяющих теорию. Обычно уравнения имеют вид $f(x_1,...,x_n) = E$, где $x_1...x_n$ - переменные или термы специального вида (образцы), а E - выражение возможно включающее f, $x_1,...,x_n$. Все переменные считаются универсально квантифицированными и уравнение или система уравнений определяет функцию f, являющуюся решением этого уравнения (системы)[9]. В функциональных языках, в отличие от логических функции служат для описания как алгоритмов, так и структур данных. Таким образом, все функциональные символы разбиваются на два класса: вычисляемых функций и "вычислительно инертных" конструкторов, хотя с чисто логической точки зрения разницы между ними и нет. Некоторые языки имеют фиксированный набор конструкторов, другие позволяют определять новые конструкторы. Иногда разделение на два класса выполняется неявно: вычисляемыми считаются те функции, для которых заданы определения.

Запросы и ответы в функциональных языках представляют собой просто замкнутые термы. Причем ответы должны быть корректны, то есть должно соблюдаться равенство $t_{вопр} = t_{отв}$. Чтобы избежать тривиальных ответов вида 2+2 = 2+2, на форму ответов накладываются дополнительные ограничения. Они должны быть в нормальной форме, то есть не должны содержать вычисляемых функций.

Функционально-логические языки объединяют функциональную форму определения теорий с логическим стилем вопросов и ответов. В качестве запросов используются термы, но уже не обязательно основные. Если терм содержит переменные, то ответами будут различные значения этого терма вместе со значениями переменных.

Конечно, многообразие выразительных средств этим не исчерпывается. Во многих функциональных языках для определения функций используются условные равенства. С другой стороны, некоторые логические языки позволяют определять функции, опять же посредством условных равенств. Практически во всех языках в той или иной степени присутствуют конструкции высших порядков.

Наверное, стоит лишний раз отметить, что разнообразные ограничения необходимы для *эффективной* реализации поиска ответов, поскольку речь идет о языках и системах *программирования*. Если же цель достижения высокой эффективности не ставится, как, например, в системах доказательства теорем или в экспертных системах, то можно использовать и более полные версии логических языков.

## Стратегии вычислений.

Теперь, установив подходящие ограничения, мы можем разрабатывать стратегию поиска ответа, устанавливая тем самым взаимосвязь между программой, запросом и порождаемым ними процессом вычислений. Эту взаимосвязь называют *операционной (процедурной) семантикой* языка программирования. Здесь особенно ярко проявляется контраст между декларативным и императивным программированием. Если для императивных языков операционная семантика более-менее очевидна, в то время как описанная выше модельная или денотационная семантика достаточно сложна, то в декларативных языках картина обратная.

Операционную семантику можно описывать на различных уровнях, начиная от общей спецификации и заканчивая детальным описанием абстрактной машины. Мы рассмотрим только основные принципы организации вычислений, описывая алгоритмы в терминах манипулирования абстрактными выражениями.

Пусть функциональная программа задана набором равенств вида $f(x_1,...,x_n) = E$. Каждое такое равенство определяет правило переписывания, которое, будучи применено к терму $f(t_1,...,t_n)$ заменяет его термом $E[t_1/x_1,...,t_n/x_n]$, то есть термом E в котором все вхождения параметров $x_1,...,x_n$ заменены на $t_1,...,t_n$.

Для вычисления значения выражения воспользуемся тем же методом, каким вычисляют сложные выражения школьники: выберем некоторое подвыражение и заменим его равным ему, но более простым. В качестве правил замены будем использовать равенства, составляющие программу. Если выражение содержит терм вида $f(t_1,...,t_n)$, а программа - подходящее уравнение, то к этому терму можно применить правило переписывания. С полученным выражением поступаем так же и т.д. пока не получим выражение в нормальной форме. Этот процесс переписывания термов называется *приведением* или *редукцией*. Подвыражения, которые могут быть переписаны, называются редексами (redex - reducible expressions приводимое выражение). Если выражение не содержит ни одного редекса, то оно не может быть более упрощено и, следовательно, находится в нормальной форме.

Алгоритм приведения можно описать следующим образом

```
пока выражение содержит редексы
    выбрать редекс
    применить к нему подстановку
    заменить редекс результатом подстановки
```

Этот алгоритм содержит некоторую степень неопределенности. Во-первых, выражение может содержать несколько редексов, во-вторых, для данного редекса может существовать несколько допустимых подстановок. Второй случай в функциональных языках обычно запрещён: определение функции должно быть детерминированным. Таким образом, порядок приведения зависит от стратегии выбора редекса. В некотором смысле выбор стратегии несущественен, поскольку он не влияет на результат в силу следующей важной теоремы.

> Если выражение приведено двумя различными способами к двум нормальным формам, то эти нормальные формы будут совпадать.

Однако выбор стратегии влияет на возможность получения результата, в чем нетрудно убедиться на примерах. Пусть заданы определения функций

```
f(x) = 6 + 4;
g(y) = y / 0;
h(x) = h(x)+1;
```

Попытаемся вычислить различными способами значения выражений (подчеркнуты редуцируемые выражения)

```
f(g(1)) -> 6 + 4 -> 10
```
или
```
f(g(1)) -> f(1/0) -> ошибка: деление на 0

f(h(1)) -> 6 + 4 -> 10
```
или
```
f(h(1)) -> f(h(1)+1) -> f(h(1)+1+1) -> f(h(1)+1+1+1))   ...
```

Если алгоритм приведения всегда выбирает самый внутренний редекс (не содержащий внутри себя других редексов), то говорят, что он применяет *аппликативный* порядок редукции. Если же выбирается самый внешний редекс (тот, который не содержится внутри других редексов), то говорят, что алгоритм применяет *нормальный* порядок редукции. Другими словами, при аппликативной стратегии все значения аргументов функции вычисляются перед ее вызовом, и если вычисление аргумента приводит к аварии или бесконечному циклу, то вычисление выражения приводит к тому же, хотя , возможно, этот аргумент и не требовался для получения результата. При нормальной стратегии аргументы передаются функции "как есть" и вычисляются только при необходимости. Важность нормальной стратегии определяется следующей теоремой.

> Если выражение можно привести к нормальной форме любым способом, то это можно сделать, применяя нормальную стратегию.

Но если нормальная стратегия так хороша, спрашивается, зачем обсуждать что-то ещё? Главная проблема: нормальная стратегия, вообще говоря, требует большего числа шагов, чем аппликативная. Например, вычислим значение f(1 + 2 + 3) для функции f определенной как

```
f(x) = x + x + x
```

сначала используя аппликативную стратегию

```
f( 1 + 2 + 3)
-> f(3 + 3)
-> f(6)
-> 6 + 6+6
-> 12+6
-> 18
```

а затем нормальную:

```
f(1+ 2 + 3)
-> (1 + 2 + 3) + (1 + 2 + 3) + (1 + 2 + 3)
-> (3 + 3) + ( 1 + 2 + 3) + (1 + 2 + 3)
-> 6 + ( 1 + 2 + 3) + (1 + 2 + 3)
-> 6 + ( 3 + 3) + (1 + 2 + 3)
-> 6 + 6 + (1 + 2 + 3)
-> 6 + 6 + (3 + 3)
-> 6 + 6 + 6
-> 12 + 6
-> 18.
```

Поскольку аргументы передаются функции не вычисленными, они должны перевычисляться всякий раз, как появляются в теле функции.

Вполне разумно вычислять аргументы только когда они потребуются. Но столь же разумно вычислять аргумент только один раз, даже если он потребуется больше чем однажды. Это может быть достигнуто оптимизацией. Выражение f(1+ 2 + 3) могло бы вычисляться как-нибудь так:

```
f(1 + 2 + 3)
-> x + x + x { x = 1 + 2 + 3}
-> x + x + x { x = 3 + 3}
-> x + x + x { x = 6}
-> 6 + 6 + 6
-> 12 + 6
-> 18
```

Этот прием "вынесения за скобки" повторных вычислений аргументов называется *ленивыми вычислениями*. Ленивые вычисления - достаточно эффективный метод реализации нормальной стратегии. В действительности, нормальная стратегия без ленивых вычислений практически не применяется, так что эти термины стали синонимами, и часто противопоставляют "ленивые" и "активные" или "энергичные" вычисления. Однако даже с такой оптимизацией нормальная стратегия менее эффективна, чем аппликативная. Причем она не только требует большего времени, но использует намного больше памяти, потому что необходимо хранить все "недовычисленные" выражения. Поэтому, очень актуальная проблема - анализ строгости аргументов функций. Если удается определить, что данный аргумент понадобится при любых обстоятельствах, то его имеет смысл вычислить перед вызовом функции. Типичный пример - условное выражение "`if C then T else F`" строго по `C` и нестрого по остальным аргументам.

Наконец, мы ведь вовсе не обязаны строго определять последовательность редукций. В самом деле, в достаточно сложном выражении всегда найдется несколько редексов и если у нас есть несколько процессоров, почему бы не проводить редукции параллельно? Даже

если результаты некоторых из этих вычислений не понадобятся и время на них окажется потраченным впустую, в целом достигается значительный выигрыш в производительности. Такая "сверхактивная" стратегия получила название "спекулятивные вычисления". Правда, на этом пути встречаются трудности. Нелегко определить, какие редукции можно выполнять одновременно. Ирония судьбы: хотя интерес к функциональным языкам в свое время был вызван их лучшей приспособленностью для параллельных вычислений, большинство работ посвящено их реализации на последовательных компьютерах. Когда-то казалось, что распараллеливание программ достигается автоматически, но оказалось, что это далеко не так, и в последнее время оживились попытки добавить в языки средства управления параллелизмом.

Тот же метод переписывания подходит и для реализации логических языков. Каждое утверждение вида A :- $B_1$,..,$B_n$ будем интерпретировать как правило переписывания A $\rightarrow$ $B_1$,...$B_n$, а в случае n=0 как "правило стирания" A $\rightarrow$ . Действительно, для того чтобы данное предложение A было выводимым, необходимо чтобы оно присутствовало в программе явно, либо как следствие одного из утверждений A :- $B_1$,..,$B_n$ и таким образом доказательство A сводиться к доказательству каждого предложения из совокупности $B_1$,...,$B_n$. Если удается стереть исходное предложение (цель), то его можно считать доказанным. Дополнительная сложность связана с тем, что в процессе переписывания происходит присваивание значений переменным.

По аналогии с редукцией термов можно предложить алгоритм для редукции целей.

```
цель := запрос
пока цель не стерта
  выбрать редуцируемую подцель
  если таких подцелей нет
    неудача
  выбрать подходящее правило подстановки
  применить правило подстановки
  заменить подцель результатом подстановки
успех
```

В отличие от предыдущего, этот алгоритм содержит двойную неопределенность. Недетерминированным оказывается не только выбор цели, но и выбор правила. И если выбор цели не влияет на результат, то выбор правила подстановки весьма существенен. Для правильной работы данный алгоритм должен угадывать верное правило подстановки. Конечно, при реализации угадывание придется заменить перебором. Если бы у нас имелась недетерминированная машина, она могла бы создавать несколько копий текущего процесса, по одной для каждого правила подстановки. Как только какой-нибудь из процессов успешно завершается, цель доказана.

При реализации на обычной детерминированной машине придется хранить все варианты целей, которые могут привести к успеху. Для этого можно завести список целей, подлежащих редукции.

```
список целей := {запрос}
пока (список целей не пуст) и (ни одна из целей не стерта)
  для всех G из списка целей
    удалить цель G из списка
    выбрать подцель S цели G
    для всех правил подстановки, применимых к S
      применить правило подстановки к S
      заменить в G подцель S результатом подстановки
```

```
        добавить полученную цель к списку целей

если список целей пуст
  неудача
иначе
  успех
```

Описанный алгоритм обычно называют стратегией поиска в ширину. Подобно нормальной стратегии, он обладает свойством полноты - если предложение может быть доказано, то доказательство может быть найдено поиском в ширину. Однако, он обладает и подобными недостатками: порождает большой объем вычислений и требует очень много памяти. Более распространенный метод реализации логических языков - перебор с возвратами или поиск в глубину. Идея этого метода - выполнять редукции последовательно, так, как будто мы угадываем нужное правило, но по ходу дела отмечать какое правило было применено. Если процесс завершается неудачей, возвращаемся к последнему месту выбора и пытаемся применить другое правило. Если же в этом месте все правила исчерпаны, возвращаемся к предпоследнему и т.д. Для организации вычислений обычно создаётся стек точек возврата, каждая из которых хранит состояние вычислений и оставшиеся правила подстановки. При откате отмечается очередное правило, а если таковых не осталось точка возврата удаляется из стека и операция повторяется для предыдущей точки.

```
пока цель не стерта
  выбрать подцель
  выбрать подходящее правило подстановки
  если такое правило есть
    отметить точку возврата
    применить правило подстановки
    заменить подцель результатом подстановки
  иначе
    если определена точка возврата
      вернуться к запомненному состоянию
    иначе
      неудача
успех
```

В Прологе и производных от его языках дополнительно уточняется порядок выбора подцелей и правил. Всегда выбирается самая первая по тексту программы подцель и первое правило. Это позволяет воспринимать его как несколько необычный процедурный язык программирования со встроенным механизмов отката.

Простейший способ реализации функционально-логических языков - преобразовать функции в отношения и таким образом перейти к логической программе. Этот нехитрый метод "расплющивания" (flattening) применяется во многих языках, но в последнее время интерес специалистов сосредоточен на специальных алгоритмах, позволяющих повысить эффективность функционально-логических программ. Наиболее многообещающим кажется метод, получивший название сужение (narrowing). Он представляет собой редукцию с тем отличием, что при подстановке допускаются присваивания значений переменным. Количество предложенных стратегий и расширений сужения довольно велико и всем им ещё предстоит пройти практическую проверку[10].

Исходя из сказанного, можно сделать вывод, что функциональная (логическая) программа представляет собой просто набор правил подстановки. Это действительно так. Тогда, спрашивается, к чему все эти разговоры о теориях и моделях? Но сила декларативного

подхода как раз и заключается в том, что можно отвлечься от процесса исполнения программы и рассуждать только в терминах описываемой модели.

---

[1] См.: А. Стругацкий, Б. Стругацкий "Сказка о Тройке".

[2] Проблеме вопросов и ответов не уделялось должного внимания. Это значительный пробел в теории, учитывая что возможности языков определяются не только средствами описания проблем, но допустимыми формами запросов.
См.: M. O'Donnell "Introduction: Logic and Logic Programming Languages".

[3] Первая цепочка записана в общепринятой математической нотации; вторая - в бесскобочной символике Яна Лукасевича, известной как "польская запись"; третья представляет собой запись на Лиспе. Наконец, последняя запись соответствует нотации, применяемой Фреге в его "Begriffsschrift" (переводилось под названиями "Шрифт понятий" и "Исчисление понятий"). Это исторически первая система обозначений для логики предикатов. Линиями обозначается импликация, а черточками - отрицание. Строго говоря, данный пример следует читать как "¬(q(a,b) => ¬p(a))", но в системе Фреге нет разницы между этим выражением и эквивалентным ему "p(a) & q(a,b)".

[4] Здесь я ловко проскочил несколько интуитивно ясных, но довольно тонких моментов. За подробностями можно обратиться к любому учебнику математической логики. Лучшими остаются классические работы

- А. Чёрч "Введение в математическую логику".
- С. Клини "Математическая логика".
- Х. Карри "Основания математической логики".
- Э. Мендельсон "Введение в математическую логику".

К сожалению, сейчас эти книги труднодоступны. Можно порекомендовать

- В.А. Петухин "Курс дискретной математики".
- V. Detlovs, K. Podnieks "Introduction to Mathematical Logic".
- Peter Suber, "Logical Systems".
- R.B. Jones "Factasia Logic".

Сравнению естественного языка и языка логики посвящена Глава 6. "Логический анализ языка" монографии В.Ф. Турчина "Феномен науки", которая стоит того, чтобы её почитать в любом случае.

[5] Эквациональные логики нас будут интересовать в связи с функциональным программированием. Однако, интересно отметить, что эта область активно разрабатывалась (Э. Дейкстрой, Д. Грисом и др.) применительно к формальным методам разработки программ.
См.: D. Gries, F. Schneider "An introduction to teaching logic as a tool".

[6] Другие разновидности "теории типов" можно найти в работах:

- J.W. Lloyd, "Higher-order computational logic" (.ps ~200k),
  in Computational Logic: From Logic Programming into the Future, Springer-Verlag, 2002.

- D. McAllester "Higher Order Logic"(.ps ~100k)
- L.C. Paulson "A formulation of the simple theory of types (for Isabelle)" (.pdf ~200k), In: COLOG-88: International Conf. in Computer Logic.Tallinn, Estonia (1988).

См. также статью Д. Миллера "Logic, Higher-order" из "Энциклопедии ИИ".

[7] Использование греческой буквы "лямбда" не несет никакого смысла. Это обозначение возникло в процессе эволюции из символа циркумфлекса (^), который сначала ставился над переменной, затем перед ней, и, наконец, превратился в лямбду. Гораздо более выразительно обозначение ($x \to t$), введённое Бурбаки и популярное у европейских математиков.

[8] Этой замечательной теореме, не имеющей, впрочем, прямого отношения к теме, посвящена книга К.М. Подниекса "Вокруг теоремы Геделя".

[9] Несколько боле слабые ограничения на форму уравнений приводят к языкам эквационального программирования - интересной, но малораспространенной парадигмы, естественно обобщающей функциональное программирование.
См.: M. O'Donnell "Equational Logic Programming".

[10] Анализ различных стратегий для функционально-логического программирования можно найти в статье:
M. Hanus "A unified computation model for functional and logic programming" (.dvi) Paris, 1997.

Обзор общей теории и применений систем переписывания дан в статье: N. Dershowitz "A Taste of Rewriting" (.ps). Более подробную информацию можно получить на сайте посвященном таким системам http://rewriting.loria.fr/.