**НАУЧНЫЕ ОСНОВЫ ДОКАЗАТЕЛЬНОГО ПРОГРАММИРОВАНИЯ**
**НАУЧНОЕ СООБЩЕНИЕ**

Современному машинному программированию свойствен эмпирический подход. ЭВМ – это автомат, который, полностью подчиняясь командам программы, демонстрирует некоторое поведение. Эмпирический характер программирования означает, что правильность программы определяется путем апостериорного (последующего за программированием) наблюдения за поведением машины, исполняющей данную программу. Этот процесс испытания получил название отладки программы и повсюду признается не только неотъемлемым, но и самым трудоемким этапом на пути ее создания.

При поверхностном взгляде на программирование это положение, казалось бы, не должно свидетельствовать о каком-либо неблагополучии. Сейчас очень популярно трактовать программирование как еще один вид инженерной деятельности, в частности подчеркивать его сходство с машиностроительным проектированием. При таком подходе стадия длительных и интенсивных испытаний опытных образцов является совершенно необходимой, при всей своей дороговизне спасающей от еще больших расходов в период эксплуатации изделия. Тем не менее, есть глубокие и актуальные причины не маскировать специфику составления программ привлекательными аналогиями, а повышать теоретический уровень программирования, добиваться гарантированного качества программ еще на стадии их разработки. Укажу лишь на две из этих причин, которые представляются наиболее существенными.

Первая причина – внешняя. Довольно долго ЭВМ использовались главным образом в научной работе, проектировании и организационно-финансовом управлении. Машины были сосредоточены в вычислительных центрах, а результаты их вычислений контролировались работой специалиста. В результате дефекты программного обеспечения были локализованы, а их проявления нейтрализовывались сравнительно быстро. Сейчас положение стремительно изменяется. Во-первых, ЭВМ начинают все в большей степени управлять полностью автоматизированными и сложными процессами большого масштаба или высокого быстродействия, в которых контролирующее вмешательство человека-оператора сведено к минимуму или полностью отсутствует. Во-вторых, распространение вычислительных систем коллективного пользования персональных ЭВМ делает партнерами вычислительных машин миллионы людей, не являющихся специалистами ни по программированию, ни по компьютерам. В обоих случаях потребитель полностью вверяет себя вычислительным машинам и заложенным в них программам. Эта возрастающая зависимость человечества от вычислительной техники резко повышает требования к надежности и достоверности программных средств.

Вторая причина необходимости повышения качества программирования скорее внутренняя. В большинстве традиционных областей применения ЭВМ возможность использовать машину возникает лишь после того, как найдена математически точная формулировка задачи и предложен строгий метод ее решения. Более того, и сама задача и метод ее решения могут быть сформулированы лишь в контексте так называемой теории предметной области, в которой математическая модель исследуемого явления сочетается с неким общематематическим знанием, работающим на эту модель. Существующая практика программирования такова, что точное знание, воплощенное в условии задачи и описании метода ее решения, не находит своего закономерного и неискаженного воплощения в программном тексте, а служит лишь справочной информацией, "сырьем" для программиста. Команды программы выписываются программистом интуитивно, в некотором смысле наугад, с наивной верой в успех. В результате, если, скажем, программисту нужно реализовать вычисление $y = sin\ x$ по формуле ряда Тейлора

$$y = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots,$$

то, получив некоторую программу и просчитав по ней для некоторого количества аргументов, он будет сличать ее результаты с доступной ему таблицей синусов и в итоге сделает эмпирическое заключение о пригодности программы. В то же время, если бы удалось доказать, что программа проводит вычисление именно по указанной формуле, то это единственное рассуждение было бы достоверней десятков расчетов.

Эта утрата исходного знания, постоянно происходящая в массовом программировании, является главной причиной неэффективности программирования и одновременно стимулирует создание его научных основ. Надо так изменять процесс программирования, чтобы любая попытка написать в программе команду, не вызванную условием задачи и применяемым методом ее решения, приводила бы к формально обнаруживаемому противоречию. В результате программирование становится доказательным.

Поиски систематических методов программирования, обладающих свойством доказательности, имеют уже достаточно длительную историю, восходящую к началу существования электронной вычислительной техники. Сейчас уже можно говорить о создании научных основ доказательного программирования, которые должны стать опорой специальной подготовки программистов и новой технологической дисциплины программирования. В эту работу внесли вклад ученые из разных стран, в частности Ф.Л. Бауэр, Р. Берсталл, Э. Дейкстра, Дж. Маккарти, М. Нива, В. Турский, Р. Флойд, А. Хоар, и советские ученые Я. М. Барздинь, В. М. Глушков, С. С. Лавров, А. А. Летичевский, А. А. Ляпунов, Э. Х. Тыугу.

*Синтезирующее программирование* [1] – это программирование в его наиболее чистом виде, то есть процесс получения программы "из ничего", исходя из условия задачи и метода ее решения. Например, если мы говорим: "Решить такую-то систему обыкновенных дифференциальных уравнений методом Рунге–Кутта", то условием задачи являются правые части дифференциальных уравнений, обозначения переменных величин и их начальные значения, а методом решения – расчетные формулы Рунге–Кутта. Доказательное программирование задач подобного рода не очень сложно: правила вычисления, по существу, определены, и их нужно лишь организовать. Чаще, однако, условие задачи и метод ее решения переплетены таким образом, что она выглядит как перечисление объектов (указывается, какие величины или функции даны, что надо определить), дополненное описанием свойств этих объектов в виде разного рода соотношений или условий, на них наложенных. В этом случае метод решения посредством некоторой систематической процедуры должен быть выбран или найден с помощью анализа перечисленных свойств и воплощен в тексте программы.

С формальной точки зрения условие задачи записывается в виде совокупности логических формул, в которые входят символы известных и неизвестных величин, операций и функций. Такая совокупность формул получила название спецификации задачи. В последующем синтезе программы по спецификациям различают два подхода: логический и аналитический (или трансформационный).

При логическом подходе спецификация трактуется как формулировка теоремы, утверждающей существование решения задачи. Синтез программы состоит в поиске доказательства этой теоремы существования в некотором конструктивном логическом исчислении. Если такое доказательство удается построить, то существуют формальные правила, позволяющие преобразовать доказательство в программу, находящую решение задачи. Существенно, что есть универсальная контролирующая процедура, которая проверяет правильность доказательства и применения правил перехода от доказательства к программе.

При аналитическом подходе спецификация трактуется как уравнение относительно программы. Оказывается, такое уравнение можно подвергать символическим преобразованиям, при которых символ неизвестной программы "рассыпается" на систему вспомогательных неизвестных, а они, в свою очередь, заменяются либо другими неизвестными, либо конкретными программными конструкциями. Таким образом, по мере преобразований синтезируемая программа постепенно "прорастает" в тексте спецификации, в конце концов превращая ее в законченный программный текст. Опять-таки правильность выполнения символических преобразований может формально проверяться на каждом шагу.

Заметим, что при каждом из указанных подходов синтез программы из спецификаций остается творческой задачей: в первом случае надо найти доказательство, во втором – правильное направление преобразований. На практике применяется комбинация обоих подходов.

Если говорить менее формально, то синтез программы – это некоторый способ манипулирования знаниями: специальным знанием, воплощенным в условии задачи, предметным, характеризующим область, в которой возникла задача, и универсальным знанием, отражающим общематематические закономерности и правила доказательного рассуждения. (Слово "манипулировать" здесь не случайно, оно подчеркивает некоторое осязаемое и точное представление знания, позволяющее проследить и проконтролировать последствия его использования.) В целом такое словоупотребление оказалось методологически весьма плодотворным, поскольку позволило объединить универсальную строгость математической логики с разнообразием предметных областей, проницаемость человеческой интуиции с безошибочной пунктуальностью машинной обработки.

В качестве примера синтезирующего программирования рассмотрим синтез программы, которая возводит вещественное $x$ в натуральную степень $n$. Условием задачи является вещественность $x$, натуральность $n$ и математическое определение степенной функции, согласно которому $x^n$ – это такая функция, что

$$x^1 = x,\ x^{n+m} = x^n \times x^m \text{ и } x^{n\times m} = (x^n)^m.$$

Сначала мы подвергаем это конкретное знание анализу, чтобы узнать, достаточно ли оно для нахождения любого значения степенной функции. В результате математического рассуждения мы выводим из исходных свойств, что $x^0 = 1$ и что можно вычислять любую положительную степень по соотношению $x^{n+1} = x^n \times x$ и любую четную степень по соотношению $x^{2n} = (x^n) \times (x^n)$. Эти соотношения получаются благодаря использованию предметного знания о свойствах натуральных чисел, а также интуитивных соображений о пользе применения знания в его простейших формах (сведение суммы к счету и произведения к удвоению). Дальнейший анализ обнаруживает, что наши способы возведения в степень перекрываются, так как $n + 1$ может быть и четным и нечетным. Легко, однако, избежать пересечения способов, употребив соответствующее соотношение только для нечетного показателя:
$x^0 = 1,\ x^{2n+1} = x^{2n} \times x,\ x^{2n} = (x^n) \times (x^n)$.

Затем вступает в силу трансформированный подход, который с учетом свойств операций счета и удвоения приводит к более явной записи:

$$x^n = \begin{cases} 1, \text{если } n = 0; \\ (x^{n/2}) \times (x^{n/2}), & \text{если } n \text{ четное}; \\ x \times x^{n-1}, \text{если } n \text{ нечетное}, \end{cases}$$

откуда уже всего один шаг до законченной и нетривиальной программы *ст (x, n) = {n = 0 |1| {чет n | ст (x, n/2) × ст (x, n/2)| x × ст (x, n – 1)}}*. В левой части равенства стоят имя определяемой функции и ее аргументы. Определение носит рекурсивный характер, то есть сводит вычисление функции к более простым случаям. В фигурных скобках стоят три члена, разделенные вертикальной чертой. Первый член – условие, второй – правило вычислений при выполнении условия, третий – правило вычислений при невыполнении условия.

Итак, основу синтезирующего программирования составляют творческие находки, которые, главным образом, определяют направление преобразования. Они опираются на ряд вспомогательных логических

[1] См. Грис Д. Наука программирования. – М.: Мир, 1984.

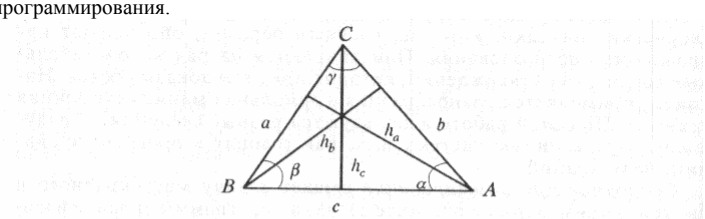
утверждений, которые при этом доказываются. Наконец, применяется разнообразная формальная манипуляционная техника. При этой работе используются разные сведения, образующие определенную систему или, как говорят в программировании, базу знаний.

*Сборочное программирование* решает задачу многократного и быстрого применения в процессе создания программы заранее изготовленных деталей.

Для того чтобы представить полезность и важность такого подхода к программированию, достаточно вспомнить о роли сборных конструкций в современном домостроении. Роль деталей в сборочном программировании играют программные модули. Это – программы, обладающие определенной структурной и функциональной целостностью и в то же время специально приспособленные к тому, чтобы вступать в четко определяемое и контролируемое информационно-логическое взаимодействие с другими модулями (под взаимодействием понимается или обмен информацией, или определенная соподчиненность выполнения).

Сборочное программирование эффективно, когда комбинирование сравнительно небольшого числа заранее запрограммированных модулей позволяет быстро решить любую задачу из некоторого класса часто возникающих проблем. Ориентация на класс задач – особенность сборочного программирования – объясняет его актуальность, поскольку широкое распространение мини- и микроЭВМ позволяет применять каждую отдельную машину для решения определенных специальных задач. В общем случае сборочное программирование – это тоже синтез программы по спецификации задачи, однако в условиях, когда отдельные блоки ее решения уже отработаны и запрограммированы. При этом синтез конкретной программы сводится к извлечению из условия задачи схемы сборки модулей, а эта работа существенно более проста, и ее легче контролировать. Мало того, для некоторых классов задач схема сборки может извлекаться из условия задачи по формальным правилам, и в результате процесс построения программы приобретает полностью автоматический характер.

В качестве примера сборочного программирования рассмотрим синтез программы, которая по заданным площади и двум высотам треугольника находит его стороны и углы. Эта задача принадлежит классу задач на решение треугольника по заданным трем его элементам. Теоретическую основу сборочного программирования составляет так называемая модель предметной области – в данном случае база в виде системы равенств I–VI, связывающих элементы треугольника. С каждым равенством можно связать несколько его разрешенных вариантов, выражающих явно одну величину через другие. Программы соответствующих формул являются модулями в данной системе сборочного программирования. Иногда разрешенные формулы могут быть получены из исходных равенств автоматически с помощью символических преобразований, а в общем случае все варианты модулей строятся методами синтезирующего программирования.




I. $\alpha + \beta + \gamma = \pi$; $\alpha = \pi - (\beta + \gamma)$

II. $\frac{\sin\alpha}{a} = \frac{\sin\beta}{b} = \frac{\sin\gamma}{c}$; $a = \frac{\sin\alpha \times b}{\sin b}$; $a = \arcsin\frac{a \times \sin\beta}{b}$

III. $a^2 = b^2 + c^2 - 2bc \times \cos\alpha$; $a = \sqrt{b^2 + c^2 - 2bc \times \cos\alpha}$; $h_a = \frac{2S}{a}$; $b = \sqrt{b^2 - c^2 + 2bc \times \cos\alpha}$

IV. $S = 1/2\, ah_a$; $a = \frac{2S}{h_a}$; $h_a = \frac{2S}{a}$

V. $S = \sqrt{p(p-a)(p-b)(p-c)}$; $p = \frac{a+b+c}{2}$

VI. $S = 1/2\, bc \times \sin\alpha$; $b = \frac{2S}{c \times \sin\alpha}$; $\alpha = \arcsin\frac{2S}{bc}$

Общая схема сборки программы для отдельной задачи выглядит следующим образом. Фиксируются известные и требуемые величины. После этого, наблюдая модель, выстраивают цепочку равенств, которая ведет от известных величин к неизвестным. Если таких цепочек оказывается несколько, с помощью дополнительных соображений выбирается одна из них. Когда цепочка не выстраивается, задача в данной модели не имеет решения. Возникающая при этом комбинаторная задача напоминает поиск пути в лабиринте, игру в домино или отгадывание головоломки. Выстроенная цепочка и есть схема сборки. Затем для каждого звена выбирается соответствующий вариант разрешенной формулы в качестве модуля собираемой программы.

Если считать, что в треугольнике заданы площадь $S$ и высота $h_a$ и $h_b$, то получится, например, следующая схема сборки:

IV. $S = 1/2ah_a$ (a по S и $h_a$);
IV. $S = 1/2bh_b$ (b по S и $h_b$);
VI. $S = 1/2ab \times \sin\gamma$ ($\gamma$ по S, a и b);
III. $c^2 = a^2 + b^2 - 2\, ab \times \cos\gamma$ ( c по a, b и $\gamma$);
VI. $S = 1/2bc \times \sin\alpha$ ($\alpha$ по S, b и c);
VI. $S = 1/2ac \times \sin\beta$ ($\beta$ по S, $\alpha$ и c).

Взяв соответствующие варианты разрешенных формул, получим программу решения треугольника:

$a = 2S/h_a$; $b = 2S/h_b$; $\gamma = \arcsin(2S/ab)$; $c = \sqrt{a^2 + b^2 - 2ab \times \cos\gamma}$;
$\alpha = \arcsin(2S/bc)$; $\beta = \arcsin(2S/ac)$.

Оставшиеся неиспользованными равенства модели можно применять для контроля вычислений, например $\alpha + \beta + \gamma = \pi$..

В настоящее время сборочное программирование – наиболее развитый практически метод доказательного программирования[2].

[2] См.: Кахро М. И., Калья А. П., Тыугу Э. Х. Инструментальная система программирования ЕС ЭВМ (ПРИЗ). – М.: Финансы и статистика, 1981.

*Конкретизирующее программирование* в некотором смысле противоположно сборочному – оно дает метод систематического построения специализированных программ из универсальной заготовки. Использованию ЭВМ для решения нового класса задач обычно предшествуют теоретическое исследование этого класса, разработка теории предметной области. Как правило, признак полноты такой теории – создание универсального метода решения любой задачи из данного класса. Такой метод гарантирует продуктивность попыток применения ЭВМ к рассматриваемому классу задач. Более того, универсальный метод может быть запрограммирован.

Пусть класс задач характеризуется некоторой функцией f от нескольких, скажем двух, аргументов x и y. Тогда пара величин a и b – это условие единичной задачи в классе, a f (a, b) – это решение задачи (a, b). Как известно, любая машинная программа $\pi$ с аргументами $\xi$, $\eta$ осуществляется с помощью воплощенного в конструкции машины универсального процесса Int выполнения (интерпретации) программы. Другими словами, если $\pi$ ($\xi$, $\eta$) вычисляет функцию $\varphi$ ($\xi$, $\eta$), то результат интерпретации программы $\pi$ с исходными данными и дает соответствующее значение функции $\varphi$: Int (p; $\xi$, $\eta$) = $\varphi$ ($\xi$, $\eta$).

На практике, однако, применение универсальных методов ограничивается тем обстоятельством, что по отношению к конкретной задаче общий метод обычно является избыточным. Всегда появляется соблазн, используя некоторые особенности в постановке задачи, найти частный, но более эффективный способ решения. С теоретической точки зрения поиск частного способа решения задачи на ЭВМ – это поиск особой программы $p_A$, рассматриваемой на некотором сужении A исходных данных x,y и вычисляющей на этом сужении ту же функцию, что и универсальная программа Int ($p_A$; x, y ) = Int (p; x, y ) = f (a, b), но делающей это вычисление более эффективно (за меньшее время или с меньшим расходом памяти). Сужение области исходных данных – это то же самое, что задание некоторой дополнительной информации об этих величинах. Максимальной информацией о данных является их конкретное задание – сужение области данных до одной точки (a, b). В этом случае специализированная программа $p_{a,b}$ вообще не содержит аргументов, а просто выдает заранее вычисленное готовое решение: $p_{a,b}$ имеет вид **"выдать** f (a, b)**"**. Такой метод таблиц готовых решений был одним из главных способов специализации универсальных методов в "домашинный" период науки и техники.

Одним из наиболее важных среди разного рода суждений является фиксация, или связывание, параметра, когда при нахождении функции f (x, y) для одного значения a переменной x приходится вычислять f (a, y) для многих значений y. Тогда можно надеяться, что специализированная программа $p_a$ (y) как функция меньшего числа переменных и как использующая информацию о значении параметра a будет работать существенно эффективней, решая для любого y = b ту же задачу (a, b): Int ($p_a$; b) = Int (p; a, b) = f (a, b).

В практике программирования постоянно происходит поиск специальных методов решения задач на основе связывания параметров. Зачастую такие поиски подчеркнуто игнорировали общую теорию в надежде получить более эффективное решение. Одно из наиболее интересных достижений теории программирования – обнаружение принципиальной возможности разрабатывать специализированные методы решения, применяя некоторую систематическую процедуру к программе универсального метода[3].

Более точно в любом языке программирования может быть построена общая процедура Mix, называемая смешанным вычислителем, которая для любой программы p (x, y ) и для любого значения a параметра x строит специализированную программу $p_a$(y). Процедура Mix имеет три группы аргументов Mix = Mix(p; x; y), где p – программа, реализующая функцию f (x, y), x и y – аргументы программы p, при этом x – связываемая часть аргументов (заданные параметры), а y – аргументы, оставшиеся свободными. Имеют место следующие свойства:

Mix (p; x, y) = p(x, y);<R>
Mix(p; a, b) = <B>выдать f(a, b);<R>
Mix(p; a; y) = p<MIV>a(y );<R>
Int(p<MIV>a; b) = Int(p; a, b) = f(a, b).

Смешанный вычислитель при отсутствии дополнительной информации об аргументах оставляет универсальную программу в неприкосновенности при полной информации об аргументах, принимая на себя всю работу, строит тривиальную программу выдачи готового решения и при частично заданной информации выдает специализированную программу, учитывающую доступную информацию. Смешанные вычисления составляют основу конкретизирующего программирования.

Для того чтобы продемонстрировать механизм смешанных вычислений, следует напомнить о так называемом редукционном способе вычисления выражений, известном нам еще из школьного опыта. Например, чтобы вычислить значение формулы решения квадратного уравнения $(b+\sqrt{b^2-4ac})/2a)$ для значения аргументов a = 3, c = 4 и b = 8, нужно заменить символы переменных их значениями, а затем выполнить цепочку редукций, состоящих в замене операции и ее аргументов результатом применения операции к этим аргументам:

$(8+\sqrt{8^2-4\times3\times4})/2\times3,$
$(8+\sqrt{64-4\times12})/6,$
$(8+\sqrt{64-48})/6,$
$(8+\sqrt{16})(8+\sqrt{16})/6,$
$(8+4)/6,$
$12/6,$
$2.$

Редукции такого типа соответствуют обычным правилам выполнения программ при полностью определенных исходных данных.

Из элементарной же математики нам известны способы частичной редукции, упрощения формулы для некоторого заданного сужения области значений ее аргументов. Например, можно рассмотреть процесс упрощения той же формулы $(b+\sqrt{b^2-4ac})/2a$ для a = 1, b = 2d, в и c – свободны:

[3] Ершов А. П. Смешанные вычисления. – В мире науки. – 1984. – № 6.

$(2d + \sqrt{(2d)^2 - 4 \times 1 \times c})/2 \times 1,$
$(2d + \sqrt{4(d^2 - 4c)})/2,$
$(2d + \sqrt{4(d^2 - c)})/2,$
$(2d + 2\sqrt{d^2 - c})/2,$
$d + \sqrt{d^2 - c}.$

В результате редукций, дополненных элементарными алгебраическими преобразованиями, получается остаточное выражение, которое является специализацией исходной общей формулы, учитывающей указанную дополнительную информацию об аргументах.

Оказывается, что аналогичную систему преобразований можно распространить и на программы. Естественно, что редукции формул должны быть дополнены правилами для выражений, содержащих условия, и для функциональной подстановки. Например, для нотации, употребленной в этой статье с целью записи программы вычисления $x^n$, эти правила имеют следующее значение.

*Замена на равное*. Если в программе есть выражение F и известно, что $F \equiv F^1$, то F можно повсюду заменить на $F^1$. Это преобразование позволяет выполнять вычисления по заданной информации.

*Редукция условия*. Если в конструкции {C |P |Q} известно значение условия C, то конструкция может быть заменена на P или Q в зависимости от истинности или ложности условия C. Это преобразование позволяет удалить программы вычисления, противоречащие имеющейся информации.

*Подстановка.* Если обе части равенства содержат некоторую переменную, то все ее вхождения могут быть заменены на произвольное выражение. Это правило – главное средство внесения более конкретной информации в программу.

Теория показывает, что существует некоторая последовательность применения этих преобразований, которая при полностью заданных аргументах вычисляет результат программы, при отсутствии заданных входных значений оставляет ее нетронутой, а при частичном задании аргументов редуцирует ее к так называемой остаточной программе, которая и становится специализированной версией универсального метода решения. Ниже в качестве примера показана последовательность преобразований общей программы вычисления $x^n$ к специальной версии, рассчитанной для вычисления $x^3$. Существенно, что этот процесс осуществляется автоматически.

Универсальная программа для $x^n$

ст(x, n) = {n = 0| 1| {чет n | ст (x, n / 2) × ст (x, n /2)| x × ст(x, n - 1) } (1)

конкретизация n = 3 и подстановка

ст(x, 3) = (3 = 0 | 1| {чет 3| ст (x, 3/2) × ст (x, 3/2) | x × ст (x, 3 – 1}}}

замены и редукция условия

ст(x, 3) = x × ст (x, 2) (2)

конкретизация n = 2 и подстановка в (1)

ст(x, 2) = {2 = 0 | 1| {чет 2| ст (x, 2/2) × ст (x, 2/2) | x × ст (x, 2 - 1)}}

замены, редукция условия и замена в (2)

ст (x, 3) = x × (ст (x, 1) × ст (x, 1)) (3)

конкретизация n = 1 и подстановка в (1)

ст(x, 1) = {1 = 0 | 1| {чет 1| ст (x, 1/2) × ст (x, 1/2) | x × ст (x, 1 - 1)}}

замены, редукция условия и замена в (3)

ст(x, 3) = x × ((x × ст (x, 0)) × (x × ст (x, 0))) (4)

конкретизация n = 0 и подстановка в (1)

ст(x, 0) = {0 = 0 | 1| {чет 0 | ст (x, 0, 2) × ст(x, 0/2) | x × ст (x, 0 - 1)}}

замены, редукция условия и замена в (4)

ст(x, 3) = x × ((x × 1) × (x × 1))

замена на равное (F × 1 ≡ F)

ст(x, 3) = x × (x × x)

Хотя за каждым из отмеченных трех подходов к доказательному программированию уже стоит определенная теория, подтвержденная конкретными применениями и экспериментами, тем не менее, мы находимся лишь в самом начале пути решения проблемы в целом. Прежде всего, нужно научиться применять эти три подхода в комбинации. Для этого необходимо создать общую концептуальную, нотационную и языковую среду, в которой происходят построение и преобразование программ. Надо иметь общий и строгий языковый баланс, в терминах которого формулируются условия задач, выражается база знаний и записываются различные формы программ и утверждения о них, выраженные в виде строгих математических конструкций. Эта среда сейчас формируется, другими словами, создается лексикон программирования[4].

В рамках лексикона программирования должна быть разработана фундаментальная теория составления программ, описывающая рассмотренные три подхода, но в виде не конкретных примеров, а общих теорем, основных соотношений и конструкций, которые, по существу, и составляют научную базу программирования. Для того чтобы применять этот единый подход к программированию самых разных задач, нужно в соответствующих предметных областях найти ту базу знаний, модельные соотношения, которые позволят достигать нужной точности и строгости в постановке задачи.

Можно сказать, что перед вычислительным делом возникает задача, аналогичная той, которая стояла перед математикой и естествознанием в XVIII–XIX вв. и которая привела к созданию математического

[4] Ершов А. П. Предварительные соображения о лексиконе программирования // Кибернетика и вычислительная техника. – М.: Наука, 1984. (См. также настоящий том, с .449.)

анализа и на его основе целой гаммы инженерных наук: технической физики, строительной механики, электротехники и др. Теперь нужно построить анализ и исчисление программ и на его основе создать инженерные дисциплины применения электронных вычислительных машин в самых разных областях человеческой деятельности.