Ордена Ленина
Институт прикладной математики
Академии наук СССР

Г.Ш. Вольдман, И.Б. Задыхайло

Некоторые соображения об определении степени непроцедурности языков программирования

Препринт № 51 за 1977 г.

Москва

Сформулировано правило сравнения степени непроцедурности запросов. На нескольких примерах показано, как работает это правило. Указаны источники повышения непроцедурности в запросах. Продемонстрирована необходимость большего использования аппарата математической логики и теории множеств в языках программирования. Подчеркнуто влияние непроцедурности запросов на устойчивость языка по отношению к языкам реализации.

> В некоторых отношениях мы напоминаем первобытного человека, который вылез из пещеры и ощутил разницу между днем и ночью: мы были счастливы от того, что улавливаем различие между процедурностью и непроцедурностью [12].

1. Введение

В настоящее время во многих работах, связанных с языками программирования, употребляются такие термины как непроцедурность языка и уровень языка. Между тем, не существует четкого определения ни того, ни другого понятия, они воспринимаются интуитивно, а зачастую смешиваются.

Под уровнем языка обычно понимают [1] лаконичность формулировки процесса решения задачи из некоторого класса. Типичными примерами языков являются система команд машины БЭСМ-6 и Алгол-60. Последний называют языком более высокого уровня, отражая тот факт, что он позволяет описывать алгоритмы из области вычислительной математики более лаконично, чем на языке машины БЭСМ-6. Это связано с тем, что в Алголе-60 определены такие структуры и операции, которые являются, с одной стороны, более емкими, а с другой стороны, лучше отвечают требованиям указанного класса задач.

Путем введения набора программ в БЭСМ-6 или накопления процедур в Алголе-60 можно построить язык еще более высокого уровня, в котором задачи из некоторого класса можно было бы формулировать с любой, зависящей только от этого класса, степенью лаконичности. Однако такое повышение уровня языка производится в рамках тех возможностей, которые заложены в том языке (базовом), над которым надстраивается данный. Более того, при таком подходе только через этот язык и может быть понята семантика вводимых понятий и алгоритма решения. Рассмотренная схема повышения уровня языка соответствует методу "снизу-вверх" построения программных систем.

В настоящее время широкое развитие получил альтернативный метод построения систем "сверху-вниз". В этом случае делается попытка сформулировать язык сначала в терминах высокого уровня, как в

отношении структур данных, так и в отношении операций, исходя из класса задач. Однако описание семантики алгоритма и соответствующих понятий тут требует постепенного уточнения от его записи в обобщенных терминах до формулировки в терминах языка, который может быть использован непосредственно для решения задачи (языка реализации).

Теперь можно подвести некоторый итог. С точки зрения уровня языка пользователь может быть полностью удовлетворен при обоих подходах. Уровень языка относится исключительно к интерфейсу программиста с программной системой и отражает степень удобства формулировки решения задач из данного класса.

Отметим следующую существенную разницу, отличающую два рассмотренных подхода. При подходе "снизу-вверх" использование алгоритма на языке высокого уровня предполагает известным и однозначным алгоритм реализации на базовом языке. При втором подходе на каждом этапе уточнения возможен выбор различных решений, т. е. в этом подходе заложена возможность поэтапной "притирки" к базовому языку.

Из этого обстоятельства можно извлечь очень важное следствие. Предположим, что заранее известен класс языков реализации, на которых может потребоваться провести решение задачи. Тогда можно, проводя поэтапное уточнение алгоритма, сначала принимать решения, не касающиеся языков реализации, и лишь затем зависимые от них решения. Таким образом повышается мобильность программной системы. Понятно, что такого рода мобильность можно использовать и для повышения эффективности алгоритма, выбирая нужные пути конкретизации в зависимости от различных целей.

Теперь мы подошли к сути проблемы. Для полного использования преимуществ, вытекающих из указанного следствия, необходимо, чтобы запись исходного алгоритма (с учетом семантики) не закрывала возможности выбора ни одного пути достижения цели. Поэтому формулировка цели должна указывать конечный результат, никак не уточняя способы его достижения. При выполнении этого условия запрос (задание цели) является полностью непроцедурным. Степень непроцедурности понижается с введением в запрос различных уточнений.

Авторы, конечно, понимают, что задача создания непроцедурных языков является чрезвычайно сложной, а точная формальная запись цели часто оказывается самой программой на базовом языке.

Однако уже существуют языки, включающие важные элементы непроцедурности (запросы), в известной степени отвечающие указанным требованиям.

Повышение уровня языка и включение в него непроцедурных элементов повышает устойчивость базового языка реализации в отношении изменения архитектурных принципов вычислительных

машин и их элементной базы.

Авторы подчеркивают это обстоятельство, так как именно исследования по выбору базового языка реализации для вычислительной машины [8] привели их к рассмотрению вопросов, поднятых в этой работе.

Ниже обсуждаются подходы к определению степени непроцедурности программных элементов и на примерах демонстрируются различные аспекты проблемы. Вся работа носит поисковый характер, поэтому авторы заранее извиняются перед читателем за недостаточно формальный метод изложения.

2. Попытки определения непроцедурных языков

Некоторое время считалось, что языки программирования делятся на процедурные и непроцедурные, грубо говоря, по следующему признаку: если язык может быть использован для записи собственного компилятора или компилятора для любого другого языка, то соответствующий язык процедурный, в противном случае — непроцедурный.

Шоу [2] первым указал, что между чистой процедурностью и полной непроцедурностью лежит целый спектр промежуточных концепций.

Олле [3] вслед за этим предложил следующее определение степени непроцедурности.

> "При определении степени непроцедурности необходимо спросить, насколько подробно абонент или программист должен указывать машине, что необходимо сделать и как это сделать. Если он в состоянии просто задавать что делать, но не должен указывать как делать, язык или соответствующая функция могут быть классифицированы как непроцедурные. Если же дополнительно к этому абонент должен указывать, как следует выполнять ту или иную операцию, т. е. он обладает полным управлением, тогда язык или соответствующая функция могут быть определены как процедурные".

Такое определение степени непроцедурности, хотя и близко к нашему интуитивному представлению, но является крайне неформальным.

При попытке придать более формальный смысл последнему определению мы прежде всего должны обратить внимание на способ указания того, "что необходимо сделать". Из предыдущего обсуждения интуитивно ясно, что такое описание должно быть строго формальным и включать, по возможности, только элементы высокого уровня (т. е. уровня содержательной постановки задачи). В дальнейшем мы будем ограничиваться (не в целях иллюстрации, а в связи с состоянием исследования проблемы), в основном, отдельными программными элементами, а описания того, что программист хочет получить, будем

называть запросами. В ряде случаев для формулировки запросов уже сейчас с успехом используются формулы исчисления предикатов первого порядка. В настоящее время, вообще, резко усилилась тенденция к большему использованию в языках программирования понятий математической логики и теории множеств. Рассматриваемые нами вопросы отражают лишь некоторые аспекты, определяющие истоки указанной тенденции.

Будем теперь считать, что при некоторых одинаковых ограничениях (начальные значения, структуры данных и набор операций) нам заданы два запроса *A* и *B*, эквивалентные с точки зрения конечного результата. Обозначим через *Z(A)* множество алгоритмов (программ-реализаций), которые можно построить для решения задачи в рамках запроса *A*, а через *Z(B)* — соответствующее множество для B.

Определение

Будем говорить, что запрос *A* имеет большую степень непроцедурности, чем запрос *B* (иначе, $A *\supset B$), если $Z(A) \supset Z(B)$.

Из данного определения вытекает

Лемма

Пусть задана произвольная функция качества программы f, принимающая большие значения на лучших программах, тогда

$$A *\supset B \Rightarrow \underset{X \in Z(A)}{SUP} f(X) \geq \underset{Y \in Z(B)}{SUP} f(Y)$$

Таким образом, запрос *A* лучше запроса *B* в том отношении, что он обеспечивает возможность не худшей его реализации, независимо от того, какой критерий качества выбрать. Например, он обеспечивает возможность не менее эффективной реализации по времени на заданной машине, или выбор не менее качественной реализации при заданном критерии качества для любой машины.

Мы хотим теперь обсудить ряд примеров.

3. Примеры, раскрывающие суть введенного определения степени непроцедурности

3.1. Пусть задан некоторый язык ассемблера *L*, не содержащий макросредств, например, автокод БЕМШ для машины БЭСМ-6 [4]. Пусть над ним надстроен аппарат макросредств и получен в результате язык *L**. Например, *L** — это автокод + макрокод [15] для БЭСМ-6 или язык ассемблера для машин ЕС ЭВМ.

Пусть далее пользователю надо написать запрос вида $G(Z_1, Z_2, ..., Z_q)$, где $Z_1, Z_2, ..., Z_q$ — параметры, каждый из которых может принимать конечное число значений, а *G* описывает некоторые действия, зависящие

от параметров. Определим критерий качества как $1/n$, где n — длина рабочей программы, отвечающей данному запросу. Ясно, что каждому допустимому набору значений параметров можно поставить в соответствие наилучшую по введенному критерию программу. Пусть написано макроопределение, которое позволяет каждой макрокоманде (запросу на L^*) поставить (через макроопределение) в соответствие наилучшую программу.

Понятно, что оставаясь в рамках L, мы можем не достичь наилучшей программы, ибо ограничены методом подпрограмм, а это означает, что мы должны оставить на динамику выполнения программы как операторы выяснения свойств набора параметров, так и заготовки программы на различные случаи,

Если запрос G соответствует вычислению функции $G(x, y) = x^y$, то известно, что можно выделить целый спектр случаев, когда не следует применять общего алгоритма. Например, при $(x \neq 0, y = 0)$ $G(x, y) = 1$, при $(y = 1)$ $G(x, y) = x$, при $(x \neq 0, y = -1)$ $G(x, y) = 1/x$ и т. д.

Общий вид формы (высокий уровень) запроса позволяет "погружать" запрос в любую внешнюю среду, что делает программу более независимой. Конечно, следует иметь в виду, что если значения параметров выясняются лишь в динамике, то наилучшим может оказаться метод подпрограмм.

Однако можно утверждать, что язык L^* является более непроцедурным, чем язык L, подразумевая под этим то, что всегда можно придумать такой запрос G и такой критерий качества, при которых множество программных реализаций запроса в L^* окажется большим и позволит сделать лучший выбор. Ясно, что этому утверждению можно придать теоремный характер, значительно обобщив постановку. Однако мы не будем этого делать в этой работе.

В заключение обсуждения этого примера отметим, что достижение большей непроцедурности опиралось здесь на учет начальных условий (параметров внешней среды) запроса. Семантика же запроса описывалась в виде макроопределений. Отсюда следует важная роль единого универсального языка для записи макроопределений над любым языком. В связи с этим, авторы подчеркивают усилия, предпринимаемые для аппаратной реализации такого языка, который, в свою очередь, содержит элементы непроцедурности, используемые при его реализации [7, 8, 16].

3.2. Пусть мы ввели в язык операцию сложения векторов P и Q размерности N с целью получения результирующего вектора R. Тогда в программе программист может использовать такую форму записи этой операции (запрос), в которой должны указываться лишь знак операции и три идентификатора векторов (мы будем здесь пренебрегать описаниями).

Например, в Алголе-60 мы можем записать нашу операцию в виде

V(P, Q, R), где *V* — знак операции (название процедуры) сложения массивов. При этом следует иметь в виду, что в теле процедуры обычно записывается цикл

$$\underline{\text{for}}\ i := 1\ \underline{\text{step}}\ 1\ \underline{\text{until}}\ N\ \underline{\text{do}}\ R[i] := P[i] + Q[i];$$

который подразумевает последовательное вычисление компонент вектора.

Между тем, в языке APL мы можем позволить себе не уточнять процесс вычисления компонент. Достаточно записать в программе оператор вида:

$$R = P + Q$$

При этом создается полная иллюзия того, что мы убили сразу двух зайцев: разрешили операцию высокого уровня и освободились от необходимости задавать семантику операции. Между тем, это совсем не так. В случае языка APL мы фактически ссылаемся на строгое описание семантики в виде тех разделов математики, которые необходимы для точного понимания операции суммирования двух векторов. Математическое описание не предполагает последовательного перебора компонент, важен лишь конечный результат. Это означает, что операции над компонентами разрешается производить в любом порядке и одновременно, что может послужить источником улучшения качества программ. Наиболее важным здесь является, конечно, возможность параллельного вычисления, так как резервы увеличения производительности связываются в настоящее время с машинами, допускающими параллельную работу устройств. Теперь очевидно, что запрос о сложении векторов имеет, грубо говоря, один и тот же уровень (во всяком случае, различие их не носит принципиальный характер), но различную степень непроцедурности. Запрос в APL ∗⊃ запроса в Алголе-60, причем это различие носит качественный характер.

Рассмотрение последнего примера проясняет и принципиальный характер языка, на котором описывается семантика. Поскольку на вычислительных машинах ставятся задачи, алгоритмы решения которых описываются прежде всего в терминах той области, к которой они принадлежат, а программы и способы оценки их качества гораздо более консервативны и тяготеют к понятиям из области машинной и вычислительной математики, теории множеств и математической логики, чрезвычайно важной является разработка единого языка для формулировки задач с наибольшей степенью непроцедурности. Интересные достижение в этом направлении имеется в связи с реляционными базами данных (см. раздел 3.5). Однако теоретико-множественный подход может оказаться полезным в гораздо более широкой области. Рассмотрим следующий пример.

3.3. Пусть мы хотим определить операцию сложения компонент вектора X размерности N, получая число SUM. Такой запрос в языке APL можно записать в виде:

$$SUM \leftarrow +/X$$

Нужно, однако, иметь в виду, что семантика этой записи в APL дается следующим образом:

$$SUM = \left(\ldots \left((X_1 + X_2) + X_3 \right) \ldots \right) + X_N) \qquad (1)$$

Отсюда сразу следует, что при выполнении этой операции нужно соблюдать предписанный порядок, т. е. она содержит элемент процедурности.

Попробуем описать семантику подобного запроса в более непроцедурной форме. Обозначим множество координат вектора X через NK. Пользуясь тем, что сумма нескольких чисел определяется через сложение двух чисел, мы можем записать семантику запроса в виде (общая форма записи запроса приведена в приложении):

$$SUM(X, NK) = \{[NK = \emptyset \ \rightarrow 0] \ X(I)_{I \in NK} + SUM(X, \ NK \setminus I)\} \quad (2)$$

Эта запись означает, что для получения суммы координат нужно выделить одну любую компоненту $(I \in NK)$ и сложить её с суммой остальных компонент, которая получается таким же образом. Так следует делать до тех пор, пока множество координат не исчерпается $(NK = \emptyset)$. Данная семантика обеспечивает большую непроцедурность нашего запроса, так как разрешает выбирать не только соседние компоненты для накопления суммы, а любые. Однако для разрешения возможности использования более параллельного исполнения операций сложения желательно добиться еще большей непроцедурности. Для этого семантику можно задавать в виде:

$$SUM(X, NK) =$$
$$\{[I \in NK \ \rightarrow X(I)] \ SUM_{I \subset NK}(X, I) + SUM(X, NK \setminus I)\} \qquad (3)$$

Эту запись следует трактовать так: сумма элементов множества X равна сумме его двух непересекающихся подмножеств I и $NK \setminus I$. Если же множество состоит из одного элемента $(I \in NK)$, то его значение и есть сумма элементов этого множества. Эта запись разрешает как последовательное суммирование, соответствующее первым двум формам семантики, так и параллельное суммирование в соответствии с произвольными разбиениями. Ясно, что непроцедурность запроса при (1) меньше, чем непроцедурность при (2), а непроцедурность запроса при (3) больше, чем при (2).

В запросах (2) и (3) мы использовали аппарат теории множеств и рекурсивное описание. Наш опыт убеждает в том, что эти два аппарата являются необходимыми средствами для создания достаточно общего языка описания семантики запросов, ориентированного на достижение большей непроцедурности.

Подчеркнем еще раз большую устойчивость запроса с большей непроцедурностью, т. е. лучшую приспособляемость по условиям реализации.

Если критерий качества состоит в экономии рабочих ячеек, то мы можем реализовывать запрос при (3) по алгоритму, определяемому (1). Этот вариант соответствует минимальному количеству рабочих ячеек.

Если критерий качества состоит в экономии времени, а вычислительная машина допускает параллельное выполнение, то следует использовать возможность распараллеливания, заложенную в (3), и отсутствующую в (1) и (2). При этом, заведя необходимое количество ячеек, мы можем очевидным образом уменьшить время выполнения с $\approx |NK|$ единиц (единица — время выполнения сложения) до $\approx \log_2 |NK|$ единиц.

Еще лучшую оценку по времени мы могли бы получить, если бы захотели учесть то обстоятельство, что машина может обладать n-арной операцией суммирования, которая записывается, например, так $\sum(A, Y)$, где A — вектор с множеством координат Y. Причем эта операция определена только, если мощность Y ($|Y|$) не больше n, где n зависит от конкретной машины и не меньше двух. Для того чтобы использовать эту операцию, мы должны изменить условие остановки рекурсии в (3) на следующее:

$$[|I| \leq n \rightarrow \Sigma(X, I)]$$

Таким образом, мы можем теперь разбивать множество X до тех пор, пока его мощность не уменьшится до n. Следует отметить, что поскольку n для каждой машины свое, то здесь происходит запрос к окружению. При такой форме запроса на суммирование компонент мы можем достичь времени выполнения $\approx \log_n |NK|$.

Конечно, эти оценки носят сугубо теоретический характер, так как они не учитывают ни параметров и структуры памяти машины, ни времени на дополнительные расходы для реализации параллельных действий.

Отметим здесь то обстоятельство, что более непроцедурный вариант запроса (3) потребовал более длинной (2) записи семантики.

3.4. Рассмотрим теперь пример, связанный с запросами на вычисление выражений. Пусть дана формула вида $x = (a + b \times c) \times (d + e \times f)$. Если опираться на математическое определение семантики этой формулы, то запрос является в максимальной степени непроцедурным, порядок выполнения действий зависит лишь от

расположения скобок и приоритетности операций. Последовательности выполнения операций для нашего примера можно записать в виде:

$$\{R1 = b \times c\,, \quad R2 = e \times f\};$$
$$\{R1 = a + R1\,, \quad R2 = d + R2\};$$
$$x = R1 \times R2$$

Здесь фигурные скобки означают возможность параллельного выполнения действий и, следовательно, возможность их перестановки, точка с запятой разделяет последовательно выполняемые действия. Операнды операций $\times$ и $+$ коммутативны. *R1*, *R2* — промежуточные переменные. Отражены не все возможные последовательности.

В языке Алгол-60 подобный запрос, формально говоря, должен реализоваться в следующей последовательности:

$$R1 = b \times c;\ \ R1 = \mathrm{a} + R1;\ \ R2 = e \times f;$$
$$R2 = d + R2;\ \ x = R1 \times R2;$$

Теперь очевидно, что запрос на Алголе-60 носит менее непроцедурный характер, запрещая параллельное выполнение и изменение порядка действий и операндов. К сожалению, подобные ограничения накладывались во многих языках программирования.

Иногда порядок вычисления выражения необходимо предписывать, исходя из требований вычислительного процесса (например, для достижения большей точности). Структура запроса здесь такова, что программист может это сделать, отразив нужный порядок действий при помощи декомпозиции формул, оставаясь в рамках одной семантики, допускающей любую степень непроцедурности. Это свойство является чрезвычайно важным, так как оно оставляет программисту возможность внесения элементов процедурности на языке высокого уровня там, где необходимо "из высших соображений" исключить влияние выбора программной реализации, определяемого, например, оптимизацией по заданному критерию качества.

Поскольку при сравнении непроцедурности запросов A и B мы ссылаемся на множество допустимых реализаций, очевидно, что введение в семантику языка формальных эквивалентных преобразований запросов может послужить повышению его непроцедурности. Например, для запросов вычисления арифметических выражений южно использовать правила вынесения за скобки, внесения в скобки, группировки, вычисление общих подвыражений и т. д. Важно, чтобы эти правила были заимствованы из той же области, что и запрос. Тогда они могут описываться средствами того же языка.

Итак, мы выделили три вида неопределенностей, которые могут быть введены в запрос для возможности выбора лучшей по данному критерию качества программной реализации без изменения смысла и уровня запроса:

- неопределенность в порядке выполнения действий,
- неопределенность в выборе варианта программной реализации, в зависимости от внешнего окружения,
- неопределенность, связанную с множеством эквивалентных преобразований запроса.

Как мы видели на примере вычисления арифметических выражений, для довольно широкой (но все же специальной) области можно добиться сочетания максимально разумного уровня запроса и максимальной непроцедурности. Тем самым язык запросов оказывается устойчивым к изменениям базового языка реализации и, в частности, к системе команд машины. Выделение таких областей и соответствующих устойчивых языков запросов открывает, по нашему мнению, очень важное направление возможности внедрения аппаратных решений. Например, можно встраивать в ЭВМ процессор для вычисления арифметических выражений. Такой процессор может по своему разумению разрешать заложенную в запросе неопределенность. Процессор может быть и виртуальным, т. е. реализовываться программно. Важно только, чтобы программисту не предоставлялись средства, основанные на выборе одного конкретного решения, ни на одной стадии работы с программой. Тогда программы будут независимы от структуры и дисциплины работы процессора.

3.5. Весьма интересным с этой точки зрения является подход, принятый при работе с реляционными базами данных. В [5] Коддом были предложены альфа-выражения, основываясь на которых можно предложить удобный язык для написания запросов к базе данных, рассматриваемой как набор отношений. Так как в данном случае нас будут интересовать только функции для получения информации, то в качестве языка запросов мы будем рассматривать сам язык альфа-формул. Характерным для этого языка является его описательный характер, т. е. в запросе описывается не способ получения результата, а сам результат. Например, пусть дано отношение *Учреждение* с атрибутами *Фамилия*, *Номер отдела* и *Зарплата*, и требуется получить фамилии служащих, работающих в 12-ом отделе и получающих зарплату больше 100. Этот запрос, выраженный при помощи альфа-формул, будет выглядеть так

$$\text{СЛУЖ} = \{u\,\text{Фамилия} : u \in \text{Учреждение} \wedge (u\,[\text{Номер отдела}] = 12) \wedge (u\,[\text{Зарплата}] > 100)\}$$

Однако Хитс [6] показал, что некоторые запросы к базе данных, рассматриваемые как информационные, нельзя описать при помощи альфа-выражений. Одним из примеров такого запроса может служить следующий.

Пусть ориентированный граф задан множеством пар своих вершин, рассматриваемым как отношение G. Предположим, что в графе

нет контуров, но все вершины, из которых не исходит ни одна дуга в другую вершину, имеют петлю. Пусть в графе выделено некоторое множество дуг U. Мы хотим сформулировать запрос, который порождает множество петель, достижимых из U. Эту задачу можно интерпретировать, например, как задачу нахождения начальников высшего уровня в пределах некоторого количества отделов для заданного списка работников или как задачу нахождения машин, для которых делаются выделенные детали. (Петли введены для удобства написания запроса и не уменьшают общности.)

Хитс предложил два естественных способа решения этой проблемы: введение рекурсивных запросов и использование включающего языка, имеющего средства для итерации. Преимуществом того или иного способа предлагалось считать наглядность. Ввиду того, что это понятие сугубо субъективное, мы попробуем сравнить эти способы с точки зрения степени их непроцедурности. Для этого постараемся в сходных терминах описать эти запросы. Очевидно, что общую процедуру можно записать следующим образом:

$$\text{ШАГ}(u) = \{v: v \in G \ \wedge (v[1] = \ u[2])\}$$

где G — имя отношения, представляющего граф, а индексы $[1]$ и $[2]$ — имена доменов, содержащих номера вершин, из которых исходят и в которые входит дуга, соответственно. Эта процедура для заданной дуги u находит множество всех дуг, исходящих из вершины, в которую она входит.

Искомое множество петель можно получить двумя "независимыми" способами. Мы можем для каждой дуги построить путь, для которого эта дуга является началом, а петля — концом. Множество петель для всех путей, начинающихся дугами множества U, и является результатом запроса.

При другом подходе можно найти для каждой дуги из U множество дуг, исходящих из вершин, в которые входят дуги из U. Затем то же проделать для полученного множества и т. д., до тех пор, пока это множество не будет состоять из одних петель. Оно и будет искомым множеством.

Для более простой записи запросов, отвечающих этим двум подходам, введем особую интерпретацию присваивания $a := B$, где a — элемент множества A, а B — множество. Будем считать, что в результате такого присваивания a заменяется на произвольный элемент из B, а остальные элементы из B объединяются с A.

$$\underline{\text{ЦИКЛ}}\ 1(U) = \{\ \underline{\text{ДЛЯ}}\ \ x \in U\ \ \underline{\text{ЦИКЛ}}$$
$$\underline{\text{ПОКА}}\ \ (x[1] \neq x[2])\ \ \underline{\text{ЦИКЛ}}\ \ x := \text{ШАГ}(x);\ \} \qquad (4)$$

$$\underline{\text{ЦИКЛ}}\ 2(U) = \{\ \underline{\text{ПОКА}}\ \ \neg(\forall x \in U\ (x[1] = x[2]))\ \ \underline{\text{ЦИКЛ}}$$
$$\underline{\text{ДЛЯ}}\ \ x \in U\ \ \underline{\text{ЦИКЛ}}\ \ x := \text{ШАГ}(x);\ \} \qquad (5)$$

Авторы извиняются перед читателем за нестрогую и непривычную "алголо-теоретико-множественную" форму, которая введена из соображений краткости, но надеются, что она не вызовет двусмысленности, поскольку действие этих конструкций описано ранее.

Этот же запрос можно выразить в рекурсивной форме в виде:

$$\text{РЕК}(U) = \{[\forall x \in U\ (x[1] = x[2]) \rightarrow U] \\ \{v: \exists x \in U \wedge v \in \text{РЕК}(\text{ШАГ}(x))\}\} \qquad (6)$$

Общий вид введенного нами рекурсивного альфа-выражения и пояснения даны в приложении. Этот запрос может выполняться и как первый (4) и как второй (5) итеративные запросы, а также допускает попеременное выполнение по путям и по элементам множества.

Таким образом, можно утверждать, что запрос (6) является более непроцедурным, чем запросы (4) и (5). То есть аппарат рекурсии дает возможность увеличивать непроцедурность запросов по сравнению с итеративным аппаратом.

Попутно отметим, что запросы (4) и (5) несопоставимы с точки зрения их непроцедурности.

3.6. В этом разделе мы хотим сослаться на два подхода к построению языков, обеспечивающих большую непроцедурность по сравнению с широко известными языками, такими, например, как Алгол-60, Фортран, Кобол, PL/1.

Прежде всего, сошлемся на работы [13, 14]. В них предложено записывать программу в виде параметрической записи. Последняя представляет собой совокупность формул-равенств, левая часть которых есть переменная, а правая — некоторая функция от переменных и индексов. При каждой такой формуле-равенстве может быть написана логическая формула. В этом случае вычисления по формуле-равенству можно проводить лишь тогда, когда логическая формула вырабатывает значение истина. Кроме этого, для каждой переменной должны быть заданы (в виде множества векторов) компоненты, значения которых известны, и компоненты, значения которых должны быть вычислены.

Порядок вычисления искомых компонент переменных задается следующим правилом. На данном этапе вычислений можно подсчитать значения тех компонент переменных, для которых необходимые аргументы либо заранее заданы, либо уже вычислены на одном из предыдущих этапов счета. При этом нужно, конечно, учитывать значение логической функции.

Заметим, что приведенное правило задает порядок вычисления значений компонент переменных неоднозначно. На каждом этапе счета может представиться возможность провести счет по нескольким формулам-равенствам, а по некоторым из них, в свою очередь, возможно вычисление нескольких значений. С точки зрения результатов счета

должно быть безразлично, какой порядок вычисления избрать. Отсюда ясно, что параметрическая запись допускает, вообще говоря, группу вычислительных процессов для решения задачи, а не задает единственный процесс, как другие входные языки.

Серьезное внимание привлекают к себе языки, основанные на таблицах решений [9, 10, 11]. Такие языки находят все большее практическое применение, особенно для задач экономического плана. Основными понятиями таких языков являются условия, индикаторы условий и действия. Рассмотрим запрос в таком языке, определяющий начисление премии работнику в зависимости от стажа, пола и возраста (пример заимствован из [9]).

Условия, действие	Индикаторы условий							
	1	2	3	4	5	6	7	8
Стаж > 10 лет	Д	Д	Д	Д	Н	Н	Н	Н
Пол = мужской	Д	Д	Н	Н	Д	Д	Н	Н
Возраст > 50 лет	Д	Н	Д	Н	Д	Н	Д	Н
Премия	90	70	80	60	0	0	0	0

Для пояснения смысла этой таблицы приведем пример эквивалентной программы на Коболе, которая получится, если просматривать таблицу сверху вниз и слева направо.

ЕСЛИ СТАЖ > 10 И ПОЛ = М И ВОЗРАСТ > 50
ПОМЕСТИТЬ 90 В ПРЕМИЯ.
ЕСЛИ СТАЖ > 10 И ПОЛ = М И ВОЗРАСТ НЕ > 50
ПОМЕСТИТЬ 70 В ПРЕМИЯ.
ЕСЛИ СТАЖ > 10 И ПОЛ НЕ = М И ВОЗРАСТ > 50
ПОМЕСТИТЬ 80 В ПРЕМИЯ.

и т.д.

Соответствующий фрагмент можно представить в параметрической записи:

$(\text{СТАЖ} > 10) \wedge (\text{ПОЛ} = \text{М}) \wedge (\text{ВОЗРАСТ} > 50)$ ПРЕМИЯ = 90
$(\text{СТАЖ} > 10) \wedge (\text{ПОЛ} = \text{М}) \wedge (\text{ВОЗРАСТ} \leq 50)$ ПРЕМИЯ = 70
$(\text{СТАЖ} > 10) \wedge (\text{ПОЛ} \neq \text{М}) \wedge (\text{ВОЗРАСТ} > 50)$ ПРЕМИЯ = 80

и т.д.

Из предыдущего ясно, что для данного запроса непроцедурность в параметрической записи и при применении таблиц решений одинакова, а в Коболе непроцедурность меньше, так как она предписывает один определенный порядок для программирования условий. Между тем, в отношении уровня запроса, в нашем понимании, лучше всех табличная форма, а в параметрической записи и в записи на Коболе уровень запроса

примерно одинаков. При всем при этом табличная запись более удобна для определенного класса задач и контингента пользователей, что и утверждается в большинстве работ, посвященных языкам, основанных на таблицах решений.

В заключение этого раздела мы отсылаем читателя к главе 11 "Непроцедурные языки" работы [9], в которой содержится настойчивый призыв к разработке языков, позволяющих передать оптимизирующему транслятору право выбора последовательности команд. Кроме того, в этой главе содержатся некоторые конкретные обоснования этого призыва.

3.7. В последнее время большое внимание в научной литературе уделяется абстрактным типам данных [17]. Это направление вскрывает еще один источник достижения большей непроцедурности за счет того, что в алгоритм вводится неопределенность структуры данных. Например, вместо того чтобы описывать в языке путь к определенному элементу структуры в виде индексов или сложных имен, апеллирующих обычно к фиксированному представлению структуры, разрешается записывать обращение к функции, которая обеспечивает выборку или запись соответствующего элемента. Тело функции может настраиваться на заданное представление структуры, так как в языке определяется лишь к которому элементу она относится, а не то, как она должна к нему "добираться". Появляется возможность варьирования представлением структур данных. Тем самым расширяется класс возможных программных реализаций.

4. Заключение

Перечислим коротко результаты, которые изложены в работе.

Прежде всего заметим, что впервые предложено правило сравнения степени непроцедурности эквивалентных запросов. На нескольких примерах показано, как работает это правило. Указаны источники повышения непроцедурности в запросах. Продемонстрирована необходимость большего использования аппарата математической логики и теории множеств в языках программирования. Особо подчеркнута роль рекурсивных описаний запросов.

Авторы подчеркивают также роль непроцедурности запросов как основу их устойчивости в отношении языков реализации. Особенно большое значение этот факт имеет при создании вычислительных машин, состоящих из функциональных процессоров, каждый из которых опирается на свой язык [8].

По-видимому, полное освоение всех аспектов непроцедурности будет означать новый шаг на пути к естественному языку.

Авторы благодарят аспиранта факультета ВМиК МГУ В.В. Соловьева и студента того же факультета Я.А. Садыхова за многочисленные обсуждения близких к тематике данной работы вопросов.

Приложение

Некоторые запросы, употребляемые в работе, опираются на следующий общий формат.

ИМЗАП $(U) = \{$ (7)

$[Q_1 \to P_1] \vee [Q_2 \to P_2] \vee \ldots \vee [Q_n \to P_n]$ (8)

$R\}$ (9)

ИМЗАП — имя данного запроса, которое идентифицирует как тело запроса, так и множество, получающееся в результате запроса.

U — параметр запроса, который может опускаться.

R — выражение, определяющее множество, которое считается результатом запроса. Например, в языках для баз данных оно является альфа-выражением со свободными переменными.

Если в R упоминается ИМЗАП, т. е. запрос является рекурсивным, то строка (8) в формате обязательна, в противном случае она отсутствует.

Эта строка интерпретируется как набор условий остановки рекурсии. Условия могут быть непересекающимися, тогда их проверка может осуществляться параллельно.

Если же некоторые условия могут удовлетворяться на одном и том же значении аргумента, то об этом должно быть специальное указание и приоритет условий убывает слева направо.

Q_i — само условие остановки. В языке для баз данных это альфа-выражение без свободных переменных.

P_i — результат выполнения запроса на том шаге рекурсии, на котором выполнилось условие Q_i. В языке для баз данных это альфа-выражение со свободными переменными.

Литература

1. P. Naur. Programming languages, natural languages, and mathematics. Comm. ACM, v. 18, N 12, 1975.
2. C.J. Shaw. What a non-programmer should know about programming language. IEEE Newsletter. January, 1967.
3. T.W. Olle. UL/1: a non-procedural language for retrieving information from data bases. IFIP Congress 68 Edinburgh, August, 1968.
4. В.С. Штаркман. АВТОКОД. Препринт ИПМ АН СССР, 1969.
5. E.F. Codd. Relational completeness of data base sublanguages. Courant computer science symposia, vol. 6, Data base systems. May 1971.
6. M.H.H. Huits. Requirements for languages in data base system. IFIP TC-2 special working conference on data base description, 1975.

7. И.Б. Задыхайло и др. О повышении эффективности символьных преобразований. Препринт ИПМ АН СССР, 1975, № 15.
8. И.Б. Задыхайло, Е.И. Котов, А.Н. Мямлин, В.К. Смирнов. Вычислительная система с внутренним языком повышенного уровня. Препринт ИПМ АН СССР, 1975, № 41.
9. Э. Хамби. Программирование таблиц решений. — М.: Мир, 1976.
10. P. Dixon. Decision tables and their application. Computer and automaton. April 1964.
11. L.A. Lombardi. A general business-oriented language based on decision expression. Comm. ACM. February 1964.
12. T.W. Olle. Data definition spectrum and procedurality spectrum in data base management systems. IFIP working conference data base management. 1974.
13. Э.З. Любимский. Вопросы автоматизации программирования. Вестн. АН СССР, 1960, № 8.
14. И.Б. Задыхайло. Организация циклического процесса для параметрической записи специального вида. ЖВМ и МФ, т. 3, № 2, 1963.
15. В.М. Михелев, В.С. Штаркман. МАКРОКОД (описание языка). Препринт ИПМ АН СССР, 1972.
16. Л.К. Эйсымонт. О возможности параллельных схем реализации одного языка для программирования задач переработки текстовой информации. УС и М, № 2, 1977, Киев, Изд. Наукова думка.
17. Data: abstraction, definition and structure. Proceedings of the SIGPLAN'76 conference, Salt Lake City, Utah, USA, March 22-24, 1976. ACM SIGPLAN notices, v. 8, N 2, 1976.