Андрей Викторович Столяров

Об одном подходе к построению универсальных языков программирования

Настоящий документ представляет собой черновую версию статьи, опубликованной в сборнике статей молодых учёных факультета ВМиК МГУ №4 в 2007 году. При цитировании просьба ссылаться на опубликованный вариант статьи, используя следующие библиографические данные:

А. В. Столяров. **Об одном подходе к построению универсальных языков программирования** // Сборник статей молодых учёных факультета ВМиК МГУ, выпуск №4, М.: Издательский отдел факультета ВМиК МГУ, 2007, стр. 135–146

Для пользователей LaTeX приводится библиографическая информация в формате `bibtex`:

```
@INCOLLECTION{stolyarov:approach2007,
        AUTHOR={{A. B. Столяров}},
        TITLE="Об одном подходе к построению
               универсальных языков программирования",
        BOOKTITLE = "Сборник статей молодых учёных факультета ВМиК МГУ",
        ISSUE = "4",
        YEAR = "2007",
        PAGES = "135--146",
        ADDRESS = "Москва",
        PUBLISHER = "Издательский отдел факультета ВМиК МГУ"
}
```

СБОРНИК СТАТЕЙ МОЛОДЫХ УЧЕНЫХ факультета ВМиК МГУ, 2007, выпуск № 4, с. 135–146

УДК 518.683.8

ОБ ОДНОМ ПОДХОДЕ К ПОСТРОЕНИЮ УНИВЕРСАЛЬНЫХ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ

© 2007 г. А. В. Столяров

avst -at- cs.msu.ru

Кафедра Алгоритмических языков

Введение. Многообразие существующих языков программирования естественным образом приводит к возникновению вопроса о возможности построения *универсального* языка, т.е. такого языка программирования, который подходил бы для реализации любого проекта, не уступая при этом другим языкам ни по каким параметрам (т.е. не оставляя технических причин для предпочтения другого языка).

Иначе говоря, вопрос в том, можно ли создать такой язык программирования, чтобы для программиста, знающего этот язык и любое количество других языков программирования, при решении любой возникающей программистской задачи выбор языка программирования был бы очевиден и всегда оказывался в пользу универсального языка.

Языки программирования традиционно делились на высокоуровневые и низкоуровневые. Следует заметить, что смысл этих терминов с течением времени менялся. Если изначально под языком низкого уровня понимался исключительно язык ассемблера, а все остальные языки программирования считались языками высокого уровня, причем о сравнении относительной «высоты уровня» двух высокоуровневых языков речь не шла, то со временем концепция уровня языка стала более гибкой. С одной стороны, этому способствовало появление языка программирования C, который, не являясь языком ассемблера, позволял, тем не менее, описывать программу в терминах, близких к машинным. С другой стороны, с появлением всё большего количества языков программирования становилась более ясной недостаточная выразительность термина «язык высокого уровня», т.к. языки, являющиеся таковыми, оказывались очевидно по-разному удалены от машинного языка и возможностей машины; так, язык Pascal имеет с возможностями машины гораздо больше общего, чем, скажем, язык Prolog или язык Lisp. Это видно хотя бы из того факта, что динамические структуры данных в языках Lisp и Prolog входят в число первичных понятий языка, тогда как в языке Pascal их следует строить в явном виде; также языки Lisp и Prolog имеют механизмы автоматической сборки мусора, каковые в низкоуровневых терминах реализуются достаточно сложным образом; наконец, и сами модели вычисления, т.е. собственно работы программы, в этих языках не имеют ничего общего с низкоуровневыми вызовами функций и машинными командами, тогда как в языке Pascal достаточно очевидны правила, по которым конструкциям языка ставятся в соответствие фрагменты машинного кода.

Наконец, в последние 10–15 лет из программистской практики оказался почти полностью вытеснен язык ассемблера, вместо которого чаще всего используется язык C. Следует заметить, что в задачах, ранее решавшихся на ассемблере, обычно практически невозможно использовать большинство языков высокого уровня; C в этом смысле явно отличается от многих других языков.

Все это сделало термин «язык высокого уровня» неадекватным для классификации языков программирования. В результате концепция уровня языка трансформировалась из бинарной (низкий–высокий) в относительную (язык L_1 имеет уровень более высокий, нежели язык L_2).

Уровень языка является важным, а в некоторых случаях и определяющим критерием при выборе. Так, существуют проблемные области, в которых применение языков достаточно высокого уровня (в частности, языков, имеющих встроенный механизм сборки мусора) оказывается невозможным. К таким проблемным областям относятся, в частности, создание систем реального времени, разработка ядер операционных систем, программирование микроконтроллеров и т.п.

В то же время существуют и такие проблемные области, в которых оказывается неоправданным (хотя и возможным) применение языков программирования сравнительно низкого уровня. К таковым относятся самые разнообразные области, от задач организации документооборота на предприятии до задач искусственного интеллекта.

При обсуждении достоинств и недостатков тех или иных языков программирования часто упоминаются *парадигмы программирования*. Отметим, что сам термин «парадигма программирования» нельзя считать устоявшимся; разные авторы вкладывают в этот термин весьма различный смысл.

Можно заметить, что парадигмы программирования обычно не являются принадлежностью конкретного языка. Рассмотрим, к примеру, парадигму функционального программирования, т.е. парадигму, в рамках которой программа воспринимается как набор взаимозависимых функций, не имеющих побочных эффектов, а выполнение программы представляется как вычисление некоторой функции при заданном значении аргументов.

Некоторые языки (например, Hope [3]) требуют работы именно в таких рамках. Выход за рамки функционального программирования в языке Hope невозможен.

Другие языки, такие как Lisp или Haskell, стимулируют применение функционального программирования, однако допускают и отступления от него в виде побочных эффектов функций, применения императивных конструкций и т.п.

Если рассмотреть такие языки, как C или Pascal, можно заметить, что эти языки *допускают* применение функционального программирования, поскольку, вообще говоря, никто не запрещает в этих языках писать функции без побочных эффектов.

Наконец, язык Фортран делает применение функционального программирования невозможным.

Вообще, некоторый язык программирования может находиться с определённой парадигмой в одном из следующих вариантов взаимоотношений:

1. язык **навязывает** примененение парадигмы. Программирование на данном языке без применения данной парадигмы категорически невозможно. Примеры: язык Smalltalk и объектно-ориентированное программирование; язык Fortran и присваивания.

2. язык **понуждает** к применению парадигмы. Программирование на данном языке без применения данной парадигмы возможно, но очень неудобно. Примеры: язык Lisp и рекурсия; язык Pascal и присваивания.

3. язык **поощряет** применение парадигмы. Программирование на данном языке без применения данной парадигмы возможно и достаточно удобно, однако при освоении данной парадигмы программист получает вознаграждение в виде резко возрастающего удобства работы. Примеры: язык C++ и механизм исключений; язык Lisp и функции высокого порядка (функционалы).

4. язык **поддерживает** применение парадигмы. Язык включает в себя специальные средства для применения данной парадигмы, расчитанные на программистов, привыкших к её применению, однако допускает и другие, в некоторых случаях более удобные варианты решения аналогичных задач. Примеры: язык C++ и макропроцессирование; язык Lisp и циклы; язык C и рекурсия.

5. язык **допускает** применение парадигмы. Язык не включает никакой специальной поддержки для данной парадигмы, однако при определенных навыках программист все еще может ее применять. Примеры: язык Pascal и функциональное программирование; язык C и обобщенное программирование; язык Pascal и объектно-ориентированное программирование.

6. язык **препятствует** применению парадигмы. Применение данной парадигмы в данном языке теоретически возможно, однако связано с затратами, делающими её применение неоправданным. Примеры: язык C и обработка исключений (возможно с помощью

`setjmp()`/`longjmp()`, но требует серьезных трудозатрат); язык Pascal и виртуальные функции.

7. язык **запрещает** применение парадигмы. В данном языке недостаточно средств для применения данной парадигмы. Примеры: Fortran и функциональное программирование; Норе и императивное программирование; Bourne Shell и объектно-ориентированное программирование.

Следует особо отметить, что определяющим фактором при обсуждении языков программирования и парадигм оказывается не техническая сторона проблемы, а мышление программиста, т.е. психологическая составляющая.

Постановка задачи. Сформулируем коротко требования, без выполнения которых язык заведомо не может претендовать на статус универсального.

- Язык должен быть полностью компилируемым. Желательно, чтобы минимальный возможный для данного языка объем библиотеки времени выполнения был строго равен нулю (именно так обстоят дела для ассемблеров).
- Язык должен быть пригоден для низкоуровневого программирования. Это означает, что
  - не должно быть таких возможностей базового вычислителя (аппаратного обеспечения), которые программа на данном языке не могла бы использовать и
  - ядро языка должно включать в себя только такие средства, реализация которых очевидна и не вызывает сомнений; более сложные средства, допускающие разнообразные способы реализации, должны быть вынесены в библиотеку с тем, чтобы позволить программисту использовать собственную реализацию.
- Язык должен как минимум *допускать* (в смысле, приведённом в предыдущем параграфе) применение всех парадигм программирования, известных на момент создания языка; с другой стороны, язык не должен по возможности *навязывать* использование тех или иных парадигм.
- Язык должен включать средства генерации абстракций достаточно высокого уровня, чтобы удовлетворить запросы программистов-практиков (в идеале должна быть предоставлена возможность генерации абстракций сколь угодно высокого уровня).

Поясним сказанное. Интерпретируемое исполнение недопустимо при программировании микроконтроллеров, в системах реального времени и некоторых других областях. Если язык непригоден к низкоуровневому программированию, нам приходится исключить его из рассмотрения при реализации таких проектов, как ядра операционных систем, системы реального времени и т.п. Если язык включает средства, допускающие различные реализации, это практически заведомо означает, что в том или ином конкретном проекте реализация, предложенная разработчиками компилятора, окажется непригодной.

В случае, если язык не допускает применение какой-либо парадигмы программирования, возможно, что именно эта парадигма будет признана наиболее удобной при реализации конкретной задачи и язык окажется отвергнут.

Наконец, при ограниченных возможностях генерации абстракций высокого уровня реализация той или иной задачи на данном языке может оказаться заведомо многократно более трудоемкой, нежели на языке более высокого уровня.

Отметим, что в настоящее время автору не известен ни один язык, удовлетворяющий всем перечисленным требованиям одновременно. Из известных языков ближе всех к свойству универсальности подошел язык C++, однако включение в этот язык встроенных средств идентификации типов во время исполнения (RTTI, runtime type identification), множественного наследования в общем виде, а также сложных и неочевидных по способу реализации средств

обработки исключительных ситуаций сделали этот язык слишком высокоуровневым. Довершило дело введение в стандарт языка библиотеки шаблонных классов STL, которая современными программистами зачастую воспринимается как часть языка (хотя и не является таковой). В итоге разработчики операционных систем от языка C++ отвернулись; в современном сообществе преобладает восприятие C++ как одного из многих языков высокого уровня.

Метод непосредственной интеграции. Метод непосредственной интеграции впервые предложен в статье [1] в качестве подхода к интеграции разнородных языковых изобразительных средств в одном проекте. Суть метода в том, что синтаксические возможности базового языка (такого как C++) используются для построения выражений, синтаксически близких конструкциям языка альтернативного, такого как Lisp. В применении к языкам Lisp и C++ метод реализован в библиотеке InteLib. Библиотека позволяет, например, записывать такие выражения:

```
(L|DEFUN, ISOMORPHIC, (L|TREE1, TREE2),
  (L|COND,
    (L|(L|ATOM, TREE1), (L|ATOM, TREE2)),
      (L|(L|ATOM, TREE2), NIL),
      (L|T, (L|AND,
        (L|ISOMORPHIC,
          (L|CAR, TREE1),
          (L|CAR, TREE2)),
        (L|ISOMORPHIC,
          (L|CDR, TREE1),
          (L|CDR, TREE2)))))
).Evaluate();
```

причем семантика такого выражения полностью соответствует семантике аналогичного текста, записанного на языке Lisp:

```
(defun isomorphic (tree1 tree2)
 (cond
   ((atom tree1) (atom tree2))
   ((atom tree2) NIL)
   (t (and
     (isomorphic
       (car tree1)
       (car tree2))
     (isomorphic
       (cdr tree1)
       (cdr tree2))))))
```

Такая возможность достигается за счет переопределения символов стандартных операций (в данном случае – запятой и вертикальной черты). Важно подчеркнуть, что с точки зрения компилятора языка C++ вышеприведённый фрагмент представляет собой обычное арифметическое выражение.

Приведённый пример показывает, насколько существенно можно расширить изобразительную мощь языка программирования (в данном случае C++) за счет использования перегрузки символов стандартных операций.

Вместе с тем, C++ исходно не предназначался для подобных применений. Возможность перекрытия операции «запятая», по словам автора языка C++ Б. Страуструпа, в его исходные намерения не входила вовсе и никакого смысла в такой возможности он не видит [4].

Из сказанного можно сделать один чрезвычайно важный вывод: **если построить язык программирования низкого уровня, имеющий при этом средства для описания абстрактных типов данных и предоставляющий возможность перекрытия символов**

операций, то при определённом уровне гибкости этой возможности можно написать библиотеки, моделирующие вычислительные модели языков-носителей альтернативных парадигм программирования, в результате чего задача построения универсального языка программирования будет решена.

Ключевыми здесь являются требования о низкоуровневой сущности гипотетического языка и о высоком уровне гибкости средств построения абстракций высокого уровня.

Попытаемся сформулировать свойства языка, который предполагается создать в качестве универсального.

Наследие языка C. Прежде всего, сам по себе этот язык необходимо сделать языком низкого уровня, подобно ANSI C. Следует отметить, что даже язык C в той его версии, которая описывается стандартом C99, является языком чрезмерно высокого уровня. Так, например, C99 допускает следующий код:

```
void f(int n) {
    int a[n];
    int b[2*n];
    int c[3*n];
    /* ... */
}
```

Если при отсутствии возможности задания размерностей массивов подобным образом всегда можно было сказать, что любое имя локальной переменной представляет собой (с низкоуровневой точки зрения) ни что иное как константное смещение относительно базы стекового фрейма, то в приведённом выше примере имя `c` — это сложное адресное выражение, зависящее от переменной `n`, и способов реализации такого кода возможно несколько, причём найти среди них наиболее оптимальный не представляется возможным.

Представляется целесообразным во избежание появления лишних проблем сделать новый язык C-подобным, но лишь до определённого предела. В силу некоторых причин, которые будут рассмотрены позднее, совместимость с языком C на уровне синтаксиса невозможно сохранить без серьёзных компромиссов на пути достижения основных задач.

С другой стороны, задача создания стандартной библиотеки является крайне трудоёмкой и бессмысленной, поскольку для языка C она уже решена. В связи с этим представляется целесообразным, убрав в языке некоторые синтаксические несообразности языка C, тем не менее сохранить модель вызовов языка C, а также его систему типов; это позволит воспользоваться функциями стандартной библиотеки языка C.

Одним из недостатков синтаксиса языка C является чрезмерная перегруженность символа «запятая». Этот символ обозначает арифметическую операцию, однако он же используется для разделения формальных параметров в заголовках функций, фактических аргументов в вызовах функций, имён переменных в описаниях, элементов в сложных инициализаторах и констант в описаниях перечислимых типов. Если в каком-либо из перечисленных контекстов символ запятой требуется использовать для обозначения операции, приходится использовать лишние скобки.

Заметим, что во всех случаях кроме описания переменных запятую можно заменить на точку с запятой, сняв, таким образом, проблему. Что касается описаний вида

```
int a, b, c;
```

то их представляется целесообразным попросту запретить. Заметим, что действующий в языке C совершенно аналогичный запрет на описание нескольких формальных параметров одного типа в заголовке функции никому не мешает. В то же время, требование вместо вышеприведённого описания использовать

```
int a;
int b;
int c;
```

позволяет устранить ещё одну проблему, которую сторонники языка C попросту не замечают, но которая часто возникает у начинающих. Допустим, требуется описать два указателя на `int`. В языке C это можно сделать, например, так:

```
int *p, *q;
```

Для человека, плохо знакомого со спецификой языка C, такая конструкция выглядит нелогично, поскольку символ `*` имеет в данном случае отношение к типу, а не к переменной. Начинающие часто предпочитают записывать объявление указателя, ставя пробел после символа `*`, а не до него, как опытные программисты на C:

```
int* p;
```

Отметим, что язык C вполне допускает такое описание. Усложнив его, получим

```
int* p, q;
```

Для начинающего программиста естественно восприятие, при котором переменная `q` должна иметь тип «указатель на `int`», тогда как на самом деле эта переменная в данном случае оказывается типа `int`. Запрет описания нескольких переменных одного типа без указания типа решает эту проблему.

Язык и библиотеки. Одним из важнейших условий, необходимых для низкоуровнего программирования, является четкое отделение библиотеки (сколь угодно стандартной) от собственно языка. В частности, компилятор низкоуровневого языка не вправе ничего знать об именах (функций, переменных и т.п.), описываемых в библиотеке. В противном случае в ситуациях, когда использование (стандартной) библиотеки по тем или иным причинам нецелесообразно, язык и его компилятор оказываются неполноценны. Между тем, такие ситуации возникают гораздо чаще, чем можно подумать: стандартная библиотека того же языка C, несмотря на её логичность и высокое качество проектирования, тем не менее не используется при написании ядер операционных систем, при программировании всевозможных микроконтроллеров и т.п.

Отметим, что принцип разграничения языка и библиотеки часто нарушается даже для языка C: в частности, пресловутый стандарт C99 указывает, что операция `sizeof`, встроенная в язык, должна возвращать значение типа `size_t`, при том что этот тип в язык не встроен и должен описываться в библиотеке.

В качестве второго условия следует назвать следующее правило: всё, что *может быть* вытеснено из языка в библиотеку, *должно быть* вытеснено в библиотеку. В крайнем случае, если в языке необходим некий механизм, то лучше предусмотреть в языке не сам этот механизм, а средства, позволяющие его описать. Так, в языках высокого уровня обычно разрешается «складывать» (конкатенировать) строки с помощью операции `+` (это так, например, для языков Pascal, Java и т.п.). В этом плане язык C++ следует признать более удачным, т.к. он, не предоставляя подобных средств в самом языке, при этом позволяет переопределить операцию `+` для произвольных пользовательских типов, что, в свою очередь, позволяет описывать строки, которые можно «складывать», но не только их: с неменьшим успехом операция `+` используется для сложения комплексных чисел, матриц или, например, полиномов, если таковые описаны в виде классов.

Третье условие касается скорее не самого разрабатываемого языка, а предполагаемой политики стандартизации. Как показывает пример языка C++, стандартизация библиотеки в составе стандарта языка может нанести непоправимый урон культуре программирования на этом языке, а также сообществу программистов, использующих данный язык. Присвоение библиотеке STL статуса составной части стандарта C++ поставило другие библиотеки аналогичного назначения в негативные условия, и, кроме того, для подавляющего большинства программистов, использующих C++, введение STL в стандарт послужило сигналом к тому, что при работе на языке C++ следует *всегда* использовать STL; между тем, использование STL резко снижает читаемость кода, многократно усложняет отладку и сопровождение программ,

давая при этом весьма сомнительный выигрыш на стадии кодирования. Обучение начинающих программистов с использованием STL приводит к тому, что студенты, завершившие обучение, попросту не понимают принципиальных отличий C++ от других языков и не представляют программирование на C++ иначе как с использованием STL.

Вместе с тем, стандартизация интерфейсов библиотек имеет ряд несомненных достоинств, прежде всего — повышение переносимости программ.

В связи с этим представляется целесообразным проводить стандартизацию библиотек *без привязки к стандарту языка*, не давая при этом никаким библиотекам статуса «стандартной библиотеки языка». Это позволит, с одной стороны, использовать положительные стороны стандартизации интерфейсов библиотек, и, с другой стороны, избежать катастрофических последствий стандартизации одной конкретной библиотеки.

Вместе с языком можно (хотя и не обязательно) стандартизовать те и только те элементы библиотеки, которые не могут быть в силу тех или иных причин переносимым образом реализованы средствами самого языка. Примером таких элементов является интерфейс для доступа к системным вызовам операционной системы.

Арифметические выражения как основа синтаксической гибкости. Чтобы сделать возможным библиотечное моделирование альтернативных вычислительных моделей (парадигм), язык должен обладать определённой синтаксической гибкостью. В то же время необходимо сохранять эту гибкость в определённых рамках: попытки создания языков программирования с полностью программируемым синтаксисом в истории известны и к положительным результатам не приводили.

Как показывает пример проекта InteLib, в действительности для моделирования изобразительных возможностей многих альтернативных языков программирования достаточно выразительной мощности арифметических выражений, при условии, что в базовом языке предусмотрен достаточно широкий спектр арифметических операций и имеются возможности их переопределения для введённых пользователем типов.

Отметим, что слова «достаточно широкий» в предыдущем абзаце могут быть истолкованы весьма по-разному в зависимости от конкретной предметной области. Так, для моделирования выражений языка Lisp оказалось достаточно операций, имеющихся в языке C++; более сложный язык, такой как Haskell, столь удачно в имеющемся наборе операций представить не удастся.

Операция «пробел». Коль скоро символам инфиксных операций выделяется столь важная роль, целесообразно будет сделать набор таких операций возможно более широким. В частности, введение «операции пробел» (то есть соглашения, при котором два выражения произвольных типов, стоящие друг за другом и не разделённые знаком операции, считаются операндами бинарной операции, называемой «операция пробел») способно резко повысить выразительную мощность арифметических выражений.

В частности, при наличии операции «пробел» обозначить применение математического оператора к его аргументу можно будет способом, напоминающим традиционный для математики: `D f`.

Приоритет такой операции следует, видимо, сделать ниже, чем приоритеты всех имеющихся в языке арифметических операций. Действительно, выражение `a+b c+d` воспринимается скорее как два выражения сложения, записанные одно за другим, нежели как сумма из трёх элементов, вторым из которых является вызов операции «пробел», в особенности если вместо пробела использовать, скажем, символ перевода строки. То же самое можно сказать и обо всех остальных операциях, исключая разве что операцию «запятая». Вопрос о соотношении приоритетов операций «запятая» и «пробел» оставим пока открытым.

Также открытым оставим и вопрос об ассоциативности операции «пробел», то есть о том, какой из её аргументов вычисляется первым, левый или правый.

Сочетание операции «пробел» с операцией «вызов функции». Действие «вызов функции» в языках C и C++ является операцией, обозначаемой как выражение, первый операнд которого представляет собой выражение типа «адрес функции» (имя функции является

примером такого выражения), а за первым операндом следуют круглые скобки, воспринимаемые как символ операции; внутри круглых скобок перечисляются фактические параметры вызова.

Круглые скобки, таким образом, играют в языке две совершенно различные роли. С одной стороны, они используются для группировки подвыражений в выражениях с целью изменения порядка применения операций. С другой стороны, круглые скобки обозначают операцию вызова функции.

В языках C и C++ это не создаёт никаких проблем, поскольку синтаксически эти ситуации полностью разнесены: если перед круглой скобкой находится законченное выражение, то скобка обозначает вызов функции, тогда как если перед скобкой никакого выражения нет либо выражение не закончено и необходим ещё один операнд (то есть скобка находится в позиции, где ожидается выражение), то скобка воспринимается как группирующий символ.

При введении в язык операции «пробел» ситуация несколько изменяется. Для случая вызова функции от нуля аргументов проблем не возникает, поскольку при использовании скобок в качестве группирующих символов закрывающая скобка не может оказаться сразу за открывающей. При условии использования для разделения аргументов символа “;”, а не запятой (вообще говоря, попросту символа, не являющегося изображением какой-либо операции), проблем не возникает также и в случае вызова функции от двух и более аргументов (вызов отличается от группировки по наличию символа, разделяющего аргументы).

Что касается случая вызова функции от ровно одного параметра, то выражение вида `a(b)` оказывается имеющим два различных трактования: как операция вызова функции `a` от аргумента `b`, либо как вызов операции «пробел» для аргументов `a` и `b` (в этом случае аргумент `b` представляется заключённым в круглые скобки с целью группировки, что вполне имеет смысл, скажем, если `b` представляет собой, в свою очередь, выражение, с операцией, имеющей более низкий приоритет, чем вызов функции).

Возможно выбрать одно из этих толкований на основе контекстных условий и дополнительных соглашений. Подобно этому в языке C++ инициализация контруктором по умолчанию, применённая при описании переменной, оказалась бы неотличима от описания функции от нуля аргументов, если бы только для этого применялись общие синтаксические правила:

```
A a(25,26);// объект класса A,
           // конструктор от двух аргументов
A a2(77);  // объект класса A,
           // конструктор от одного аргумента
A a3();    // прототип функции от 0 аргументов,
           // которая возвращает объект класса A
A a4;      // объект класса A, использован
           // конструктор от нуля аргументов
```

Заметим, что описание отдельно стоящей переменной — это единственный случай, когда конструктор от нуля аргументов вызывается без круглых скобок. Так, в выражении

```
( A(27, 44) + A() )
```

имеет место создание двух анонимных объектов класса `A`, первый из которых создаётся конструктором от двух аргументов, второй — конструктором от нуля аргументов. Заметим, что скобки в данном случае на месте, а выражение

```
( A(27, 44) + A )
```

является синтаксической ошибкой.

Следует отметить, что решение, применённое в C++, нарушает концептуальную целостность синтаксиса языка (т.к. вводит особый случай), обладает определённой неочевидностью и затрудняет восприятие синтаксиса, в особенности для начинающих программистов. В связи с этим для решения проблемы неоднозначности между вызовом функции от одного аргумента и вызовом операции «пробел» предлагается несколько иное решение, основанное на общем правиле без особых случаев и обладающее дополнительными полезными свойствами.

Кортежи. Под кортежем будем понимать синтаксический конструкт языка, состоящий из нескольких выражений произвольных типов, разделённых символом «точка с запятой» и заключенных в круглые скобки, например:

```
(25; "a string"; x+y)
```

Выделим два особых случая, а именно, кортеж, состоящий из нуля элементов и обозначаемый `()`, а также кортеж из одного элемента, который положим семантически эквивалентным самому этому элементу (таким образом, произвольное выражение становится частным случаем кортежа, а именно, кортежем из одного элемента). Введём соглашение, что при записи кортежа из одного элемента скобки можно опустить; таким образом, например, выражения `25` и `(25)` останутся эквивалентными, как и до введения понятия кортежа.

Будем считать, что кортеж с указанием количества и типов параметров является, в свою очередь, типом данных с точки зрения профилей функций (то есть может, например, выступать в качестве параметра функции), но при этом не является, вообще говоря, структурой данных, то есть не может быть присвоен переменной (как не может и описываться переменная типа «кортеж»). Вызов функции от кортежа из n элементов физически реализуется как вызов функции от n параметров.

Логично разрешить присваивание кортежа выражений кортежу переменных, например:

```
int x;
const char *p;
float f;
(x; p; f) = (25, "string", 3.7);
```

Это позволит пользоваться кортежами имён формальных параметров при описании и вызове функций от кортежей. Например:

```
void f((int a ; int b) ; int c)
{ /* ... */ }
                  /* ... */
f((2 ; 3) ; 4);
```

Наконец, определим понятие операции вызова функции как частный случай операции «пробел» от имени функции (или, если угодно, указателя на функцию) и кортежа параметров. Как следствие, вызов функции от одного параметра станет возможно записать без скобок, например, запись `sin(x)` окажется семантически эквивалентна записи `sin x`.

Подчеркнём ещё раз, что вызовы `f(1;2;3;4)` и `f((1;2);(3;4))` можно различать с точки зрения профилей функций (т.е. компилятор должен вызывать при этом разные функции), но с точки зрения реализации ничем, кроме адреса вызываемой функции, эти два вызова различаться не должны.

Введение новых символов инфиксных операций. Как показал пример библиотеки InteLib, имеющихся в языке символов инфиксных операций может и не хватить, либо может ощущаться их недостаток.

Известны языки программирования, в которых допускается введение новых (не предусмотренных в языке) символов инфиксных операций; так, это возможно в языке Prolog.

Вместе с тем, введение операции «пробел» даёт возможность моделировать новые символы инфиксных операций, вводя их как идентификаторы объектов, для которых переопределена операция «пробел»: так, например, выражение `a in b` можно рассматривать не как операцию `in` от аргументов `a` и `b`, а как два вызова операции «пробел»; в этом случае слово `in` должно быть именем переменной, имеющей некоторый пользовательский тип. В зависимости от ассоциативности операции «пробел» это будет, соответственно, выражение

```
operator space(a, operator space(in, b))
```

или

```
operator space(operator space(a, in), b)
```

Вместе с тем, такой способ моделирования новых инфиксных операций обладает сравнительно низкой гибкостью, т.к. ограничен фиксированным приоритетом и направлением ассоциативности операции «пробел». Кроме того, в качестве символов таких операций можно будет использовать исключительно идентификаторы языка.

Как уже говорилось, к гибкости синтаксиса языка следует относится с определённой осторожностью, поэтому следует оставить открытым вопрос о целесообразности введения в язык возможности определения новых символов операций, подобной имеющейся в языке Prolog. Вернуться к этому вопросу можно будет после того, как язык будет реализован и будет получен определённый опыт работы с ним.

Наследование. Наследование как таковое является важнейшим средством создания высокоуровневых абстракций, так что необходимость наличия наследования в языке как таковая несомненна. Вместе с тем, вопросы, связанные с множественным наследованием, оказываются достаточно дискуссионными; кроме того, механизм виртуальных функций оказывается достаточно нетривиален, чтобы его нельзя было считать средством низкого уровня.

В частности, реализация *множественного наследования* в общем виде, подобно тому, как это сделано в языке C++, несмотря на кажущуюся простоту, влечёт массу нетривиальных проблем.

Действительно, пока все базовые классы неполиморфны (то есть не содержат виртуальных функций) и не являются сами чьими-то наследниками, реализация множественного наследования остаётся достаточно простой. Единственная привносимая проблема — необходимость пересчёта численного значения указателя или ссылки при преобразованиях от типа «потомок» к типу «предок» (и обратно), если соответствующий предок не является первым в списке базовых классов.

Появление у базовых классов общих предков влечёт появление понятия виртуального класса и множества сомнительных ситуаций, как в случае, если один и тот же класс является виртуальным предком части базовых классов и невиртуальным предком другой части базовых классов.

Появление в двух и более базовых классах таблиц виртуальных функций приводит к резкому усложнению реализации самого механизма виртуальных функций: для каждой виртуальной функции теперь, помимо адреса, требуются ещё, как минимум, смещения для пересчёта указателя `this` и положения указателя на таблицу виртуальных методов, так что вместо одного поля (адреса функции) строка таблицы виртуальных методов оказывается состоящей из трёх полей.

В этой связи представляется разумным компромисс, принятый в языке Java: среди базовых классов может быть не более одного класса, не являющегося *интерфейсом* (то есть классом, состоящим исключительно из чисто виртуальных функций).

В то же время статические преобразования указателей сами по себе не сложны в реализации и вполне укладываются в представления о низкоуровневом программировании, так что (при условии успешного вытеснения механизмов виртуализации в библиотеку) в язык без ущерба его низкоуровневости можно внести и более общие виды множественного наследования. Конечно, при этом следует постулировать, что наследование от нескольких классов, имеющих общий базовый класс, *всегда* приводит к наличию в объекте соответствующего количества экземпляров этого базового класса, поскольку имеющийся в языке C++ механизм виртуальных базовых классов заведомо чрезмерно сложен для низкоуровневого языка.

Библиотечная поддержка виртуальных функций. Как отмечалось выше, механизм виртуальных функций (в особенности в ситуации множественного наследования) оказывается чрезмерно сложен, чтобы считаться средством низкого уровня.

С другой стороны, библиотечная реализация виртуальных вызовов практически неизбежно потянет за собой необходимость достаточно развитых макросредств.

Само по себе это, возможно, и не столь плохо. Дело в том, что большинство возражений против применения макропроцессора, высказываемых относительно языков C и C++, основаны на том, что макроимена не подчиняются обычным для языка соглашениям об именах. Вместе с тем, язык C++ вводит т.н. шаблоны, которые отличаются от макросов практически лишь в том, что являются полноценной частью языка, подчиняются всем законам локализации имён и поддерживают проверку типов параметров (ограничение на возможные типы параметров, присутствующее в C++, выглядит скорее реализаторским, чем концептуальным).

Можно привести и другие примеры макросистем, никоим образом не нарушающих концептуальную целостность языка и не создающих никаких проблем. Такова, например, система макросредств в языке Common Lisp.

Отметим еще один момент, непосредственно связанный с виртуальными вызовами. Вообще говоря, виртуальный вызов представляет собой действие совершенно иное, нежели обыкновенный вызов. Возможно, имеет смысл зарезервировать для этого действия отдельные синтаксические соглашения (например, ввести тот или иной символ операции).

При наличии в языке макроподобных механизмов, допускающих перегрузку по типам параметров, возможно также ввести и описание инфиксных операций не (или не только) в виде функций, как это сделано в C++, но и в виде макросов (так, в C++ пришлось вводить весьма нетривиальные правила переопределения операции `->` как одноместной; определение её как макроса окажется заведомо проще для восприятия, поскольку такой макрос можно сделать двуместным). Введя возможность описания инфиксной операции в виде макроса, мы получаем возможность реализовать в виде макросов всё связанное с механизмом виртуальных вызовов, вытеснив, таким образом, эти (заведомо высокоуровневые) средства в библиотеку.

Обработка исключений. Обработка исключительных ситуаций представляет собой крайне важный механизм, использование которого повышает читабельность и надёжность программного кода и существенно снижает трудоёмкость программирования. В особенности это средство полезно в сочетании с автоматическим вызовом деструкторов локальных объектов, как это сделано в языке C++.

С другой стороны, обработка исключений в том виде, в котором она присутствует в современных языках программирования (в том числе и в C++), является средством заведомо высокого уровня. Реализация исключений неочевидна и непрозрачна.

Проблема вытеснения механизмов обработки исключительных ситуаций из ядра языка в библиотеку достаточно интересна. Ключ к решению видится в выработке удобного и логичного интерфейса к явной манипуляции стековыми фреймами; отметим, что это средство должно быть частью языка, т.к. должно позволять, например, при принудительном уничтожении стекового фрейма отработать деструкторы уже сконструированных объектов. Таким образом, механизма, подобного библиотечным функциям `setjmp()` и `longjmp()`, для этого не достаточно.

Вполне возможно, что соответствующее средство языка может быть похожим на «альтернативное тело», подобное блокам `catch` в языке C++; вместо `try`-блока следует, по-видимому, предусмотреть конструкцию, семантика которой сводится к «пометке фрейма» (для такой пометки можно использовать, например, нетипизированный указатель), и операцию «раскрутки стека на один фрейм», которая возвращала бы для помеченных фреймов значение метки, а для фреймов, не помеченных меткой — нулевой указатель.

Наличие таких средств в языке позволит предоставить программисту выбор из нескольких библиотек, обслуживающих обработку исключительных ситуаций и предоставляющих сервис различного уровня сложности. Так, не во всяком проекте требуется возможность использования *произвольного* объекта в качестве «носителя» исключительной ситуации; в некоторых случаях можно обойтись простым кодом ошибки либо фиксированной структурой данных для хранения информации об ошибке, получив при этом выигрыш в эффективности.

Следует отметить, что автору известны случаи сознательного отказа от работы с исключениями, мотивируемые чрезмерной сложностью этого механизма в языке C++. Наличие простой и прозрачной основы для такого механизма потенциально способно исправить ситуацию.

Заключение. Традиционное восприятие роли языка программирования в вычислитель-

ной системе, состоящей из аппаратуры, операционной системы и систем программирования, можно описать приблизительно следующим образом. Имеется некий конгломерат аппаратуры и операционной системы (либо даже попросту виртуальная машина), именуемый обычно *платформой* или *операционной средой*; в рамках этой среды имеется несколько *систем программирования*, каждая из которых построена вокруг того или иного языка программирования или даже некоторой конкретной его реализации. Среди имеющихся (доступных для данной платформы) систем программирования обычно для конкретного проекта выбирается какая-то одна (реже — несколько), причем выбор может быть обусловлен как поставленной задачей, так и личными предпочтениями разработчиков.

Разработка и реализация универсального языка программирования в соответствии с подходом, описываемом в настоящей статье, может несколько изменить представление о взаимоотношениях системы программирования и операционной платформы. Если сформулированные цели будут достигнуты, для заданной платформы может оказаться разумным поддерживать *один* (универсальный) язык программирования, снабженный при этом широкой коллекцией библиотек. Именно многообразие этих библиотек и займёт в этом случае нишу нынешнего многообразия языков программирования; от особенностей задачи и личных предпочтений программистов будет тогда зависеть не выбор языка, а выбор набора библиотек.

Необходимо признать, что две программы, написанные на одном и том же (универсальном) языке, но с использованием принципиально различных библиотек, могут различаться по стилю практически столь же сильно, как сейчас различаются программы, написанные на разных языках программирования. Тем не менее, использование именно одного языка с множеством библиотек вместо традиционного множества языков имеет как минимум одно важное достоинство: трудности интеграции между собой фрагментов программы, написанных в различном стиле, окажутся раз и навсегда сняты именно в силу использования одного языка.

СПИСОК ЛИТЕРАТУРЫ

1. *E. Bolshakova and A. Stolyarov.* Building functional techniques into an object-oriented system. // Knowledge-Based Software Engineering. Proceedings of the 4th JCKBSE, volume 62 of Frontiers in Artificial Intelligence and Applications, pages 101–106, Brno, Czech Republic, September 2000. IOS Press, Amsterdam.
2. *И. Г. Головин, А. В. Столяров.* Объектно-ориентированный подход к мультипарадигмальному программированию. // Вестник МГУ, сер. 15 (ВМиК), №1, 2002 г., стр. 46–50.
3. *A. J. Field and P. G. Harrison.* Functional Programming. Addison-Wesley, Reading, Massachusets, 1988. *Русский перевод: А.Филд, П.Харрисон. Функциональное программирование. М.:Мир, 1993.*
4. *B. Stroustrup.* The design and evolution of C++. Addison-Wesley, Reading, Massachusets, 1994. *Русский перевод: Бьерн Страуструп, Дизайн и эволюция языка C++, М.: ДМК, 2000.*

В статье рассматривается проблема построения универсального языка программирования. Приводится классификация языков программирования по (1) уровню абстрагирования и (2) поддерживаемым парадигмам программирования. Предлагается решение, при котором реализуется язык программирования низкого уровня (подобный языку C), снабженный развитыми средствами построения абстракций высокого уровня и соответствующими библиотеками.

Библиография 4 названия.