УДК 519.712

DOI: 10.21779/2542-0321-2019-34-2-63-71

А.М. Магомедов¹, С.А. Лавренченко², З.И. Ибрагимова¹

Алгоритм автоматизации создания тестов

¹ Дагестанский государственный университет; Россия, 367001, г. Махачкала, ул. М. Гаджиева, 43а; tagomedtagir1@yandex.ru, zaripat.ibragimova@inbox.ru

² Российский государственный университет туризма и сервиса; Россия, 141221, Московская обл., Пушкинский р-н, дп Черкизово, ул. Главная, 99; lawrencenko@hotmail.com

Создание тестов для студентов первых курсов вуза по основам программирования является длительным и трудоемким процессом. По оценкам специалистов, на подготовку качественных итоговых тестов профессиональные разработчики тратят около двух лет. Обычно такие тесты создаются творческими коллективами преподавателей под руководством признанных научно-исследовательских центров, имеющих многолетний опыт и профессионально занимающихся вопросами оценки качества образования и подготовки IT-специалистов.

Если целесообразность тотального компьютерного тестирования справедливо считается в преподавательской среде спорной, то роль тестов в качестве одного из инструментов обеспечения независимости и объективности контроля усвоения учебного материала не вызывает сомнений. Важно и то, что наличие электронных средств оценивания знаний студентов считается одним из необходимых условий для успешной аккредитации вуза. Разработка «вручную» качественных и многочисленных тестовых единиц по каждой учебной дисциплине (в среднем по 500 тестовых единиц по каждой дисциплине), причем в пяти формах, – практически невыполнимая задача, если, например, на кафедре всего восемь штатных преподавателей, а количество учебных дисциплин превышает 80.

Предложен алгоритм компьютерного формирования тестовых заданий по основам программирования на языке *Delphi 7.0*. По каждой теме учебной дисциплины рассмотрены пять формализованных структур тестовых заданий и одна «нестандартная» форма, предусматривающая творческий анализ той или иной нестандартной ситуации, возникающей при программировании средствами *Delphi 7.0*. В первой части статьи подробно изложен способ воплощения формализованных структур тестовых заданий в программное обеспечение. Ряд коллизий, возникающих при программировании на языке *Delphi 7.0*, рассмотрен во второй части статьи. Эти коллизии используются в качестве материала для составления тестовых заданий нестандартной формы. Для каждой коллизии получены исчерпывающие разъяснения путем анализа двоичных представлений в памяти значений переменных разных типов.

Результаты исследования находят применение для автоматической генерации практически неограниченного количества тестовых пунктов для компьютерного тестирования по основам языков программирования.

Ключевые слова: *тестирование, компьютер, Delphi, программирование, двоичная система.*

Введение

Данное исследование базируется на результатах, полученных в различных областях дискретной математики с применением программного обеспечения, реализующего авторские алгоритмы (см., например, [1–6]), и посвящена построению алгоритма автоматизации процесса создания тестовых единиц по языку программирования *Delphi 7.0*

[7–9] – сформулирован алгоритм построения тестовых единиц в каждой из нескольких общепринятых форм и приведены основные фрагменты воплощения алгоритма [10]. С методическими правилами проведения тестирования и с вопросами формы и содержания тестовых заданий можно ознакомиться, например, в [11].

В качестве основы для построения тестовых единиц в *нестандартной форме* рассмотрен ряд нестандартных ситуаций, возникающих в практике программирования. Описание некоторых из приведенных коллизий встречается в литературе впервые, остальные в той или иной степени узнаваемы для программиста. Во всех случаях показано, что основой для возникновения коллизии является недостаточное внимание к отображению в памяти двоичных представлений переменных того или иного типа.

Структура программного кода формирования тестовой единицы

Количество тем по основам языка программирования будем считать приблизительно равным 20. По каждой i -й теме генерируются тестовые единицы в каждой из j форм (« i, j -термы»), $j = 1, \dots, 5$:

1) выбор одного ответа из нескольких предложенных, 2) выбор двух ответов из нескольких предложенных, 3) упорядочение предложенных ответов по некоторому признаку, 4) установление соответствия между двумя множествами, 5) вычисление ответа. Дополнительно к этим формам создается некоторое количество тестовых единиц, посвященных нестандартным ситуациям, возникающим при изучении основ языков программирования.

В первой части программного кода для генерации i, j -терма объявляются и инициализируются основные переменные-объекты (главным образом – тех типов, которые изучаются в i -й теме), а также вспомогательные переменные, среди которых, в частности, могут присутствовать переменные строкового типа, предназначенные для запоминания результатов (назовем их *r-объектами*). Инициализация производится в виде присвоения одиночных констант (например, 8, 'A', FALSE, []) или константных выражений (например, not 7, trunc(-6.5), 'A'+B', random (2)).

Для рассматриваемого ниже случая генерации тестовых единиц по теме «Множества» в форме 3 (15,3-термы), объявления переменных, например, могут иметь вид:

```
var
s: array[1..5] of set of syte;
rz: array[1..5] of string;
m: array [1..5] of byte;
a, k: byte;
OutString1, OutString2: string;
Пример инициализации объектов.
for k:=1 to 5 do begin
    s[k]:=[ ];
    m[k]:= 1+ random (5); //мощности множеств
    while SetCount (s[k])< m[k] do
        //SetCount (q) – функция вычисления мощности множества q
        include (s[k], random (20));
    end;
    a:=2+random (5);
```

Затем над объектами выполняются действия, проверка понимания которых, т. е. интерпретация действий и успешная бескомпьютерная имитация их выполнения, и со-

ставляет цель опроса по данному i -, j -терму. Результаты таких действий запоминаются в r -объектах в строковом формате.

Пример блока действий:

```
s[1]:=s[1]*s[5]; s[1]:=s[1]+s[2]; s[3]:=s[4]-s[1];
if a in s[5] then include (s[4], a) else exclude (s[4], a);
for k:=1 to 5 do rz[k]:= SetString (s[k]);
// SetString(q) – пользовательская функция для представления
// множества q в строковом формате.
```

Во второй части программного кода i -, j -терма формируются две текстовые величины: открытая часть *OutString1*, непосредственно предъявляемая испытуемому, и скрытая часть *OutString2*, предъявляемая испытуемому лишь после специального преобразования программным обеспечением, осуществляющим обработку созданных тестовых единиц и организующим интерфейс с испытуемым.

Пример формирования *OutString1*:

```
OutString1:=
'№ Вопрос3'+#13#10+
'Пусть a='+intToStr(a)+ ' и над множествами ';
for k:=1 to 5 do OutString1:=OutString1+'s['+intToStr(k)+']='+rz[k]+';
OutString1:=OutString1+ #13#10+
'выполнены следующие действия:'+#13#10+
's[1]:= s[1]*s[5]; s[1]:= s[1]+s[2]; s[3]:= s[4]-s[1];'+#13#10+
'if a in s[5] then include (s[4], a) else exclude (s[4], a).'+#13#10+
'Требуется выписать количество элементов в каждом множестве, начиная с s[1].';
```

Пример формирования *OutString2*:

```
OutString2:= "
for k:=1 to 5 do
begin
OutString2:=OutString2+'№ да'+#13#10+IntToStr(SetCount(s[k]));
if k<5 then OutString2:=OutString2+#13#10;
end.
```

В конце второй части осуществляется вывод обоих текстов в файл, соответствующий *модулю*, – относительно самодостаточной совокупности из нескольких тем, включая и рассматриваемую i -ю тему.

В конце следующих двух примеров тестовой единицы, выведенной программой в файл, текстовые элементы, содержащие специальный знак «№», выписаны в одну строку для экономии места:

№Вопрос3

Пусть $a = 5$ и над множествами $s[1] = [7, 8, 17]$, $s[2] = [7, 8, 17]$, $s[3] = [6]$, $s[4] = [6]$, $s[5] = [0, 15]$ выполнены следующие действия:

```
s[1]:= s[1]*s[5]; s[1]:= s[1]+s[2]; s[3]:= s[4]-s[1];
if a in s[5] then include (s[4], a) else exclude (s[4], a).
```

Требуется выписать количество элементов в каждом множестве, начиная с $s[1]$.

№ да 3 № да 3 № да 1 № да 1 № да 2

№Вопрос2

Укажите два числа, каждое из которых представляется в памяти в виде:

10110100:

№ да – 76 № да 180 № нет 183 № нет 187 № нет – 77

Заметим, что в соответствии с требованиями программного обеспечения, используемого, например, в ФГБОУ ВО «Дагестанский государственный университет» для обработки тестовых единиц и реализации интерфейса с испытуемыми, каждая тестовая единица начинается с ключевого слова «№ Вопрос» с последующей цифрой от 1 до 5 – номером формы тестовой единицы; каждый элемент ответа предваряется служебным словом «№ да» или «№ нет», при этом только в формах 1–2 данные служебные слова соответствуют элементам ответа по смыслу: «верный» или «ошибочный», в формах же 3–4 служебное слово «№ да» применяется в качестве префикса во всех элементах ответа, обозначая их начало.

Если приведенный описательный алгоритм компьютерной генерации тестовой единицы 15,3 (тема «Множества», форма 3) позволяет, как мы надеемся, получить вполне определенное представление о структуре и способе генерации тестовых единиц и по другим темам, независимо от выбора одной из пяти перечисленных выше форм, то вопрос о нестандартной форме тестовых единиц остается пока открытым. Следующий раздел посвящен примерам нетривиальных для *начинающих программистов* ситуаций, которые легли в основу разработки нескольких тестовых единиц нестандартной формы.

Казусы «простых» арифметических выражений

В ситуациях, когда поведение целочисленных типов представляется на первый взгляд необъяснимым, как правило, оказывается достаточным подробное рассмотрение внутреннего формата их записи. Так, интересно, что следующая программа ничего не напечатает, на стадии компиляции будет сформулировано сообщение об ошибке:

```
program Project1;
{$APPTYPE CONSOLE}
var a: byte=1;
begin
  try a:= not 1; write (a) except write (0) end; readln;
end.
```

В то же время следующая программа, полученная из предыдущей элементарной заменой *not 1* на *not a*, напечатает значение 254:

```
var a: byte=1;
begin
  try a:= not a; write (a) except write (0) end; readln;
end.
```

Объяснение: при вычислении *not 1* компилятор рассматривает 1 (тогда и *not 1*) как значение переменной типа *shortInt*; следовательно, результат вычисления будет представлен в памяти как 11111110, что является дополнительным кодом отрицательного значения –2 переменной типа *shortInt*; поэтому вполне естественна диагностика ошибки при попытке присвоить отрицательное число –2 переменной *a* типа *byte*. Во втором случае вычисление *not a* выполняется вполне корректно, в ячейку переменной типа *byte* при *a=1* оператором *a:= not a* будет записано значение 254.

Рассмотрим несколько иную ситуацию.

<pre>var correct: real = 10000000000*10*0; begin writeln (correct); end.</pre>	<pre>var error: real = 10000000000*10*0; begin writeln (error); end.</pre>
--	--

Программа, приведенная слева, корректно выводит значение переменной *correct*, равное 0. А программа, приведенная справа, при компиляции выводит сообщение о переполнении при вычислении произведения чисел 10000000000 и 10, в то время как аналогичное переполнение не происходит при большем значении первого сомножителя (100000000000 вместо 10000000000), допускает простое объяснение. Прежде всего отметим, что программа

```
program Project1;
{$APPTYPE CONSOLE}
begin
  writeln (SizeOf (1000000000000)); //8
  writeln (SizeOf (100000000000)); //4
end.
```

выводит значения 8 и 4, следовательно, константы записываются в 8-байтовую и 4-байтовую ячейки соответственно. При выполнении арифметической операции над двумя целочисленными константами компилятор выделяет им в памяти две ячейки одного минимального целого типа, вмещающего каждую из этих двух констант; важно, что ячейка такого же типа выделяется и для записи результата. Следовательно, в первом случае – 64-битная ячейка целого типа (*int64*), во втором случае – 32-битная. Но ячейка типа *int64* вмещает и результат (100 млрд) произведения 100000000000*10, тогда как 32-битная ячейка не вмещает результат (10 млрд) аналогичного произведения 10000000000*10.

С упомянутым здесь типом *int64* связано следующее наблюдение. Попытка компиляции программы:

```
uses SysUtils;
var
  i: int64=0; a: real;
begin
  a:= sin(i);
end.
```

приводит к диагностике несовместимости типов.

Аналогичная ситуация сохраняется и при замене функции *sin* на *cos*, *sqrt* и некоторые другие (но не при замене на функцию *sqr*). Сказанное относится к среде *Delphi 7.0*. В то же время приведенная программа успешно компилируется, например, в версии *Embarcadero Delphi XE*. Поэтому ограничимся предположением, что казус объясняется отсутствием типа *int64* в ранних версиях *Delphi* (тип *int64* появился в 4-й версии *Delphi* в 1998 г.).

Внимательное отношение к двоичному представлению значений переменных в памяти актуально не только для целых, но и для вещественных типов. Ограничимся рассмотрением двух вещественных типов (и то в сокращенном варианте). Напомним, что двоичное представление в памяти значения *s* типа *single* состоит из трех полей, составляющих в сумме 32 бита и расположенных «от старших к младшим»: *z* (знак), *n* (смещенный порядок), *f* (мантисса) емкостью в 1, 8 и 23 бита соответственно. В случае, когда не все биты второго поля равны между собой,

$$s = (-1)^z \cdot 2^{n-127} \cdot 1.f. \quad (1)$$

У типа *extended* имеются не только количественные особенности. Двоичное представление в памяти значения *e* типа *extended* состоит из четырех полей, составляющих в сумме 80 битов, расположенных также «от старших разрядов к младшим»: *z* (знак), *n*

(смещенный порядок), один-единственный разряд, равный 1 (нормализованное представление), и f (мантисса) емкостью в 1, 15, 1 и 63 бита соответственно. В случае, когда не все биты второго поля одинаковы,

$$e = (-1)^z \cdot 2^{e-32767} \cdot 1.f. \quad (2)$$

Необходимо учитывать и то обстоятельство, что все промежуточные действия над вещественными типами выполняются после предварительного преобразования в формат *extended*. Без учета, что процессоры *Intel* оперируют с типом *extended*, нельзя объяснить, например, почему выводом следующей программы (назовем ее для определенности «программой сравнения») будет *TRUE*.

```
var
r: single=0.2; s: single=0.5;
begin
  writeln ((r>0.2) and (s=0.5)); readln;
end;
```

Покажем, что результатом сравнения $r \diamond 0.2$ является *TRUE* (равенство *TRUE* результата сравнения $s = 0.5$ доказывается аналогично). Оба операнда сравнения $r \diamond 0.2$ сначала преобразуются компилятором в тип *extended*. Для подробного анализа процесса сравнения r и 0.2 используем функции *SinToStr* ($s1$) и *ExtToStr* ($e1$) преобразования двоичного представления в памяти переменных $s1$ и $e1$ типа *single* и *extended* соответственно в их «видимые» строковые эквиваленты (составленные из нулей и единиц).

Рассмотрим программу, в начале которой предполагаются объявления функций *SinToStr* и *ExtToStr* (их реализации см. ниже).

```

...
s1: single=0.2; e1: extended; e2: extended=0.2;
begin
writeln (SinToStr (s1)); e1:=s1; writeln (ExtToStr (e1)); writeln (ExtToStr (e2));
end.

```

Используем вывод программы:

0 01111100 10011001100110011001101 (для s1),
 0 011111111111100 110011001100110011001101
 000 (для e1) и
 0 011111111111100 110011001100110011001100
 1100110011001100110011001100110011001101 (для e2)

для анализа трех двоичных представлений в памяти при выполнении вышеприведенной «программы сравнения»:

а) 0 01111100 10011001100110011001101 – значение, которое находится в переменной r типа *single* из левой части сравнения $r > 0.2$ до преобразования в тип *extended*;

б) 0 011111111111100 110011001100110011001101
00

– значение в промежуточной переменной типа *extended*, в которую преобразовано значение из *r*;

B) 0 011111111111100 110011001100110011001100
110011001100110011001100110011001101

- представление в памяти значения 0.2 из правой части сравнения $r \diamond 0.2$, после ее записи в *другую* промежуточную переменную типа *extended*.

Различие между представлениями б) и в) визуально очевидно. Используя формулы (1) и (2), нетрудно убедиться, что а) и б) суть двоичные представления значений,

отличающихся от 0.2 на мизерные значения. Не задерживаясь на этом упражнении, перейдем к отложенной ранее реализации функций *SinToStr* и *ExtToStr*.

```
type
  byte4=array [0..3] of byte; byte10=array [0..9] of byte;
function SinToStr (s1: single): string;
var
  i,j: byte; s: byte4;
begin
  result:=""; s:=byte4(s1);
  for j:=3 downto 0 do for i:=7 downto 0 do
    result:=result+ intToStr( (s[j] shr i) and 1);
  end;
function ExtToStr (e1: extended): string;
var
  i,j: byte; e: byte10;
begin
  result:=""; e:=byte10(e1);
  for j:=9 downto 0 do for i:=7 downto 0 do
    result:=result+ intToStr( (e[j] shr i) and 1);
  end;
```

Для вывода двоичного представления вещественной переменной в памяти применением операций для целочисленных типов можно предложить совмещение данной вещественной переменной в памяти с набором переменных типа *byte*. Для такого совмещения используют приведение типов (как сделано выше), директиву компилятора *absolute* или вариантную часть записи.

Все рассмотренные в данном разделе примеры, включая и реализации функций *SinToStr* и *ExtToStr*, рассматриваются нами прежде всего как материал для формулировки тестовых единиц нестандартной формы, и уже во вторую очередь – как интересные сюжеты программирования. В этом плане определенный интерес представляет и следующий пример, который мы приводим без подробных объяснений, так как лаконичных комментариев в конце строк (значений, выводимых соответствующими операторами) вполне достаточно.

```
var
  a: byte=7;
begin
  writeln (-10 mod 3 + - 10 mod - 3); //-2
  writeln (not 7);          //-8
  writeln (not - 1);        // 0
  writeln (not a: 4);        // 248
  a:= not a; write (a:4);     // 248
  // a:=not 7;              // ошибка компиляции
  readln;
end.
```

Заключение

Алгоритм воплощен в компьютерную программу, которая в течение нескольких лет используется для автоматизации разработки практически неограниченного количества разнообразных тестовых единиц по основным темам языка *Delphi 7.0* и для дости-

жения надежности тестового материала, а также для проведения независимого промежуточного контроля усвоения материала.

Работа написана при финансовой поддержке ОМИ ДНЦ РАН.

Литература

1. *Giara K.* Compact task scheduling on dedicated processors with no waiting period (in Polish) // PhD thesis, Technical University of Gdansk, IETI Faculty. – Gdansk, 1999.
2. *Petrosyan P.A., Khachatryan H.H.* Interval non-edge-colorable bipartite graphs and multigraphs // *Journal of Graph Theory*. – 2014. – Vol. 76, Issue 3. – P. 200–216.
3. *Khachatryan H.H., Mamikonyan T.* On Interval edge-colorings of bipartite graphs of small order // Cornell University Library, arXiv.org>cs>arXiv:1508.02851 (Submitted on 12 Aug. 2015).
4. *Магомедов А.М.* Цепочечные структуры в задачах о расписаниях // *Прикладная дискретная математика*. – 2016. – № 3 (33). – С. 67–77.
5. *Шихиев Ш.Б., Мирзабеков Я.М.* Дискретный анализ в синтаксическом анализе // *Информатика и её применение*. – 2018. 12:2. – С. 98–104.
6. *Шихиев Ш.Б., Мирзабеков Я.М.* Формальная грамматика русского языка в примерах // *Прикладная дискретная математика*. – 2018. – № 40. – С. 114–126.
7. *Бобровский С.И.* Delphi 7. Учебный курс. – СПб.: Питер. 2008. – 736 с.
8. *Дарахвелидзе П.Г., Марков Е.П.* Программирование в Delphi 7. – СПб.: БХВ-Петербург, 2003. – 784 с.
9. *Фленов М.Е.* Библия Delphi. – СПб.: БХВ-Петербург, 2011. – 686 с.
10. *Магомедов А.М.* Свидетельство № 2018612561 от 20 февраля 2018 г. о государственной регистрации программы для ЭВМ «Автоматизация генерации тестовых пунктов по теме “Множества в языке Delphi”».
11. *Аванесов В.С.* История педагогической теории измерений // *Педагогические измерения*. – 2012. – № 3. – С. 3–26.

Поступила в редакцию 18 апреля 2019 г.

UDC 519.712

DOI: 10.21779/2542-0321-2019-34-2-63-71

Algorithm for automating test creation

A.M. Magomedov, S. Lavrenchenko, Z.I. Ibragimova

¹ *Dagestan State University; Russia, 367001, Makhachkala, M. Gadzhiev st., 43a; magomedtagirl@yandex.ru, zaripat.ibragimova@inbox.ru*

² *Russian State University of Tourism and Service; Russia, 141221, Moscow region, Pushkino district, Cherkizovo, Glavnaya st., 99; lavrencenko@hotmail.com*

The creation of tests for first-year students on the basics of programming is a long and time-consuming process; according to experts, professional developers spend about two years to prepare high-quality final tests. Such tests are usually carried out by creative teams of teachers under the guidance of recognized research centers with many years of experience and professionally engaged in the assessment of the quality of education and training of IT-specialists.

While the questions of the expediency of total computer testing are considered controversial in the teaching environment, the role of tests as one of the tools to ensure the independence and objectivity of the control of learning material is not in doubt. It is also important that the availability of electronic means of assessing students' knowledge is considered one of the necessary conditions for successful accreditation of the University. Development of high-quality and numerous tests manually for each subject (an average of 500 test units for each subject), and – in five forms seems almost unbearable in case there are only eight full-time teachers at a Department and the number of academic subjects exceeds 80.

The algorithm of the computer forming the tests on the basics of the programming language Delphi 7.0. is presented in the article. For each theme five formal structures of test tasks and one additional form (case) are examined, providing creative analysis of a particular non-standard situation that occurs when programming using Delphi 7.0. In the first part of the article the method of implementing formalized structures of test items in the software is described in detail. In the second part of the article the incidents arising from the programming with language Delphi are discusses. These incidents are used as material for the compilation of test items of irregular shape. Full explanations for each incident by analyzing the binary representation in memory of the values of different types variables is received.

The results of this paper may find application for automatic generation of an unlimited number of test items for computer-based testing for the fundamentals of programming languages.

Keywords: *testing, computer, Delphi, programming, binary system.*

Received 18 April, 2019