

## С/к Введение в современное программирование (v.5.5)

Физфак МГУ. 2006/7 уч. год.

### Лекция 25

### Функции одной веществ. переменной

Обсудим возможную организацию работы с функциями, подчеркивая новые возможности.

**NB** Важно различать:

- процедуры-функции в языке как конструктивное средство;
- функции как объекты моделирования.

В последнем случае функции могут иметь целый ряд свойств (область определения; производные; информация о регулярности, монотонности, нулях, особенностях...), которые невозможно представить, если моделировать их процедурами и процедурными переменными.

Типичные задачи:

- изобразить график функции -- для этого нужно построить ее интерполяцию ломаными или кривыми Безье и т.п.;
- нахождение нулей, максимумов;
- решение уравнений -- обращение (ошибки);
- интегрирование (точное/быстрое/...)
- дифференцирование (при рисовании графика; при интегрировании и т.п.);
- моделирование плотности распределения по случайной выборке;
- вычисление функций (подгонка параметров -- напр., в статистической обработке эксп. данных)
- ...

Также как для комплексных чисел трудно дать универсальную реализацию, тем более для функций.

Задачи с особыми требованиями (напр., в отношении эффективности) будут требовать особых реализаций/библиотек.

Попытаемся понять некий сценарий, ориентированный на удобство использования (гибкость) и надежность.

*Удобство и гибкость всегда означают некоторую потерю эффективности,*

- но:
- 0) с таким ЯП как Оберон/КП -- можно себе позволить;
- 1) потери меньше, чем кажется на первый взгляд.

### Разминка: функции и процедурные типы

```
MODULE Kurs2006ProcedureTypes;
```

```
  IMPORT Math, StdLog;
```

```
  TYPE Function = PROCEDURE ( y: REAL ): REAL;
```

```
  PROCEDURE Integral ( a, b: REAL; f: Function ): REAL;
```

```
    CONST N = 10000000;
```

```
    VAR i: INTEGER; dx, res: REAL;
```

```
BEGIN
```

```
  dx := ( b - a ) / N;
```

```
  res := 0;
```

```
  FOR i := 0 TO N - 1 DO
```

```
    res := res + f( a + dx * ( i + 0.5 ) )
```

```
  END;
```

```
  RETURN res / N
```

```
END Integral;
```

```
PROCEDURE Do*;
```

```
  VAR res: REAL;
```

```
BEGIN
```

```
  res := Integral( 0, 2 * Math.Pi(), Math.Cos );
```

```
  StdLog.Real( res )
```

```
END Do;
```

```
END !Kurs2006ProcedureTypes.Do
```

Math.Cos = конкретное константное значение для процедурной переменной (f).  
Просто.

```
"Kurs2006ProcedureTypes" 200 0
```

```
Kurs2006ProcedureTypes unloaded
```

```
-7.933183540425812E-17
```

Почти 0, как и хотелось.

### Но:

Функции без каких-то параметров или свойств встречаются достаточно редко;  
пример -- фитирование данных в обработке эксперимента = подборка параметров так, чтобы функция "прошла по точкам"

Встречается утверждение, что ОО не нужно в численных приложениях — ЧУШЬ!  
довольно типичная ошибка в логике:

в старых библиотеках естественно таких средств нет,  
но это не значит, что "численные приложения" = "приложения, использующие старые библиотеки".

Даже в простых случаях есть несомненный выигрыш в читабельности (след., в эффективности работы) из-за организации данных и соотв. процедур.

Из документации (F1, **Component Pascal** [What's New?](#))

```
...
```

```
Procedure types
```

```
Procedure types are less flexible than objects with methods.
```

```
Even standard examples for procedure types in numerical software can benefit from modeling them as objects.
```

```
Objects are extensible, procedure types are not.
```

```
Procedure types can pose considerable implementation difficulties concerning the safe unloading of code.
```

```
For these reasons, procedure types are considered as obsolete.
```

```
For the time being, they are retained for backward compatibility
```

```
...
```

```

105
106 implementation difficulties = Как при выгрузке модуля бегать по памяти и
107 искать все процедурные переменные, которые ссылаются на выгружаемый
108 модуль?
109
110 Теперь легче понять следующее.
111
112 MODULE Kurs2006Functions;
113     IMPORT Stores;
114
115     TYPE
116         Function* = POINTER TO ABSTRACT RECORD END;
117
118     PROCEDURE ( f: Function ) Y* ( x: REAL ): REAL, NEW, ABSTRACT;
119
120 END Kurs2006Functions.
121
122 Уже можно писать процедуры и решать задачи.
123
124 повторяем:
125 Как это примерно представить себе:
126 у каждого типа есть дескриптор = спец. запись (компилятор ведь написан на
127 самом КП!), где хранится всякая информация, в т.ч. среди прочего:
128
129     Descriptor [for Function] = RECORD
130         parent: Descriptor;
131         abstract, ...: BOOLEAN;
132         size: INTEGER;
133         methods: <array> OF
134             RECORD
135                 <сигнатура>,
136                 <адрес выполняемого кода>
137             END
138     END
139     methods[0]: Y, PROCEDURE ( x: REAL ): REAL; <здесь хранятся NIL>
140
141 Компилятор требует, чтобы при конкретном расширении типа все абстрактные
142 методы были замещены конкретными.
143
144 Создавая дескриптор для конкретного потомка типа Function, компилятор
145 скопирует в соотв. поле адрес конкретной процедуры Y — причем у этого поля
146 будет точно такое же смещение относительно начала дескриптора, как и у
147 дескриптора для ***Functions.Function. Поэтому во время вызова метода
148 нужный адрес находится "в одно касание".
149 Это позволяет устроить язык и компилятор так, чтобы можно было писать и
150 правильно компилировать код, написанный под интерфейс
151 ***Functions.Function.
152
153 NB Для новых методов будут добавляться новые ячейки в дескрипторе.
154
155 Что должен сделать клиент, определяя конкретный тип-потомок для Function?
156 Пример Определить конкретную функцию-квадратный трехчлен с производной
157 и т.п.

```

```

158
159 MODULE Kurs2006FunctionsEx0;
160     IMPORT Log := StdLog, Math, In := FVTsysIn,
161         F := Kurs2006Functions;
162
163     TYPE
164         Quad = POINTER TO RECORD ( F.Function )
165             a, b, c: REAL
166         END;
167
168     (* пример конкретной функции: *)
169
170     PROCEDURE NewQuad ( a, b, c: REAL ): Quad; (* фабричная функция *)
171         VAR res: Quad;
172     BEGIN
173         NEW( res ); res.a := a; res.b := b; res.c := c;
174         RETURN res
175     END NewQuad;
176
177     PROCEDURE ( q: Quad ) Y ( x: REAL ): REAL; (* реализация метода Y *)
178     BEGIN RETURN q.a*x*x + q.b*x + q.c
179     END Y;
180
181     (* примеры процедур для любых функций: *)
182
183     PROCEDURE Integral ( f: F.Function; a, b: REAL; N: INTEGER ): REAL;
184         VAR res, dx: REAL; i: INTEGER;
185     BEGIN (* asserts ... *)
186         res := 0; dx := ( b - a ) / N;
187         FOR i := 0 TO N - 1 DO
188             res := res + f.Y( a + dx * ( i + 0.5 ) )
189         END;
190         RETURN res * dx
191     END Integral;
192
193     PROCEDURE Deriv ( f: F.Function; x, dx: REAL ): REAL;
194         VAR res: REAL;
195     BEGIN
196         RETURN ( f.Y( x + dx/2 ) - f.Y( x - dx/2 ) ) / dx
197     END Deriv;
198
199     PROCEDURE Zero ( f: F.Function; a, b, eps: REAL; limit: INTEGER;
200         OUT done: BOOLEAN; OUT res: REAL );
201     (* алгоритм деления пополам — "охота на льва в пустыне" *)
202         VAR med: REAL; fa, fb, fmed: REAL; count: INTEGER;
203     BEGIN
204         ASSERT( f # NIL, 20 ); ASSERT( ( eps > 0 ) & ( a < b ), 21 );
205         ASSERT( limit > 0, 22 );
206
207         count := 0;
208         fa := f.Y( a ); fb := f.Y( b ); ASSERT( fa * fb < 0, 40 );
209         WHILE ( ABS( b - a ) > eps ) & ~( count = limit ) DO (* по схеме поиска *)
210             (* очередной шаг: *)

```

```

211 med := ( a + b ) / 2; fmed := f.Y( med );
212 IF fmed * fb < 0 THEN a := med; fa := fmed
213 ELSEIF fmed * fb > 0 THEN b := med; fb := fmed
214 ELSE
215 a := med; fa := fmed; b := med; fb := fmed
216 END;
217 INC( count );
218 Log.Int( count ); Log.Ln;
219 Log.Real( a ); Log.Real( fa ); Log.Ln;
220 Log.Real( b ); Log.Real( fb ); Log.Ln;
221 END;
222 done := ~( ( b - a ) > eps ); (* неуспех поиска *)
223 res := ( a + b ) / 2
224 END Zero;
225
226 PROCEDURE Do*;
227 VAR f: Quad; d, i, z: REAL; done: BOOLEAN;
228 BEGIN
229 f := NewQuad( 1, -2, 1 ); (* (1-x)^2 *)
230 d := Deriv( f, 1, 1.0E-6 ); Log.String('d ='); Log.Real( d ); Log.Ln;
231 i := Integral( f, 0, 1, 10000 ); Log.String('i ='); Log.Real( i ); Log.Ln;
232
233 f := NewQuad( 0.1, 1, -0.5 ); (* 0.1 + x - 0.5 * x^2 -- ноль вблизи 0.5 *)
234 Zero( f, 0, 1, 1.0E-3, 100, done, z );
235 IF done THEN
236 Log.String('zero at z =');
237 Log.Real( z ); Log.Real( f.Y( z ) ); Log.Ln;
238 ELSE
239 Log.String('zero not found'); Log.Ln;
240 END;
241 END Do;
242
243 END !Kurs2006FunctionsEx0.Do
244 Kurs2006FunctionsEx0 unloaded
245 compiling "Kurs2006FunctionsEx0" 1208 0
246 Kurs2006FunctionsEx0 unloaded
247 d = 0.0
248 i = 0.3333333324999999
249 1
250 0.0 -0.5
251 0.5 0.025
252 2
253 0.25 -0.24375
254 0.5 0.025
255 3
256 0.375 -0.1109375
257 0.5 0.025
258 4
259 0.4375 -0.043359375
260 0.5 0.025
261 5
262 0.46875 -0.009277343749999998
263 0.5 0.025

```

```

264 6
265 0.46875 -0.009277343749999998
266 0.484375 0.007836914062500002
267 7
268 0.4765625 -7.263183593749987E-4
269 0.484375 0.007836914062500002
270 8
271 0.4765625 -7.263183593749987E-4
272 0.48046875 0.003553771972656251
273 9
274 0.4765625 -7.263183593749987E-4
275 0.478515625 0.001413345336914064
276 10
277 0.4765625 -7.263183593749987E-4
278 0.4775390625 3.434181213378919E-4
279 zero at z = 0.47705078125 -1.914739608764636E-4
280
281 Все сходится.
282
283 Что происходит:
284 • на вход Zero подается некий конкретный объект (f # NIL), имеющий некий
285 конкретный тип, являющийся потомком типа ***.Function (например, Quad);
286 раз есть такой объект, значит, был и конкретный тип — иначе компилятор бы
287 не позволил применить NEW для создания объекта, и нельзя было бы получить f
288 # NIL;
289 • в этом конкретном типе обязательно определена процедура Y (иначе
290 компилятор не скомпилирует);
291 • при обращении к процедуре Y будет вызываться конкретная процедура,
292 определенная именно для этого конкретного типа-потомка.
293
294 Др. словами, ABSTRACT-методы играют роль описателей типов процедурных
295 переменных.
296
297 Подробнее о механизме реализации (не обязательно это точно понимать):
298 § на вход Zero подается некий конкретный объект, имеющий некий конкретный тип,
299 являющийся потомком типа ***.Function; разумеется, компилятор способен сгенерировать
300 команды, "достающие" дескриптор этого конкретного типа-потомка.
301 § компилятор знает, в каком месте дескриптора типа ***.Function (т.е. на каких смещениях
302 от начала) хранится адрес процедуры Y.
303 § компилятор генерирует код, вызывающий процедуру, адрес которой хранится по
304 известному смещению (как в дескрипторе для ***.Function) в дескрипторе для конкретного
305 типа-потомка (Quad).
306 § процедура, получающая конкретный объект на место ABSTRACT-параметра будет
307 правильно работать даже если она была написана и скомпилирована задолго до того, как
308 был определен конкретный расширяющий тип и создан соотв. объект.
309
310 Замечания:
311 0) Для некоторых специальных задач алгоритм Zero не годится. В общем случае
312 следовало бы искать не значение, а интервал (см. Калиткин).
313 1) о вещах типа непрерывности и монотонности нужно делать предположения
314 или добавлять проверяющие процедуры (в любом случае полезно в
315 приложениях)
316 или (NB) вводить какие-то протоколы общения с функциями, чтобы они могли
317 "сообщать" о своих свойствах.
318

```

319 **Итого:** мы можем наделать и навставлять в Functions множество полезных  
320 процедур, которые ничего не знают о том, как вычисляются функции, с  
321 которыми они работают — но точно знают интерфейс.

322  
323 Когда параметру *f* в Zero присваивается значение конкретного типа-потомка  
324 (одного из многих возможных), а затем автоматически выбирается для  
325 исполнения правильная процедура *Y*, то это, напоминаю, **полиморфизм**.

### 326 Упражнения

327 Построить (приличные) алгоритмы:

328 0) нахождения минимума заданной непрерывной функции на заданном сегменте  
329 разными методами:

330 по правилу золотого сечения: `PROCEDURE MinGolden ( f: Function; a, b, eps:`  
331 `REAL ): REAL;`

332 по правилу парабол: `PROCEDURE MinParabolic ( f: Function; a, b, eps: REAL ):`  
333 `REAL;`

334 1) интегрирования непрерывной функции, например, используя римановы  
335 суммы и уточнение по Эйткену (см. Калиткин):

336 2) вычисления производной в данной точке с уточнением по Эйткену:

337 3) вычисления для заданной функции массива заданной длины *N*, состоящего  
338 из пар (аргумент, значение), причем аргументы распределены равномерно по  
339 области определения (которая считается компактной):

340 `TYPE Node = RECORD x, y: REAL END; Nodes = POINTER TO ARRAY OF Node;`  
341 `PROCEDURE NodesArg( f: Function; N: INTEGER ): Nodes;`

342 4) то же для равномерно распределенных значений функции (предполагая, что  
343 функция монотонно возрастает):

344 `PROCEDURE NodesVal( f: Function; N: INTEGER ): Nodes;`

345 5) вычисление интерполяционного полинома Ньютона для заданного массива  
346 *Nodes*:

347 `PROCEDURE Newton( n: Nodes; x: REAL ): REAL;`

348  
349 А теперь следим за руками:

350  
351 `TYPE`  
352 `Derivative = POINTER TO RECORD ( F.Function )`  
353 `f: F.Function`  
354 `END;`

355  
356 `Primitive = POINTER TO RECORD ( F.Function )`  
357 `(* primitive = первообразная *)`  
358 `f: F.Function`  
359 `END;`

360  
361 `.....`  
362 `PROCEDURE NewDerivative ( f: F.Function ): F.Function;`  
363 `VAR res: F.Function; d: Derivative; q: Quad;`  
364 `BEGIN`  
365 `ASSERT( f # NIL );`  
366 `WITH f: Primitive DO`  
367 `res := f.f`  
368 `| f: Quad DO`  
369 `NEW( q ); q.a := 0; q.b := f.a * 2; q.c := f.b;`  
370 `res := q`  
371 `ELSE`

372 `NEW( d ); d.f := f;`  
373 `res := d`

374 `END;`  
375 `RETURN res`  
376 `END NewDerivative;`

377  
378 `PROCEDURE ( d: Derivative ) Y ( x: REAL ): REAL;`  
379 `CONST dx = 1.0E-6;`  
380 `BEGIN`  
381 `RETURN Deriv( d.f, x, dx )`  
382 `END Y;`

383  
384  
385 `PROCEDURE NewPrimitive ( f: F.Function ): F.Function;`  
386 `VAR p: Primitive; res: F.Function; q: Quad;`  
387 `BEGIN`

388 `ASSERT( f # NIL );`  
389 `WITH f: Derivative DO`  
390 `res := f.f`  
391 `| f: Quad DO`  
392 `IF f.a = 0 THEN`  
393 `NEW( q ); q.a := f.b / 2; q.b := f.c; q.c := 0;`  
394 `res := q`  
395 `ELSE`  
396 `NEW( p ); p.f := f;`  
397 `res := p`  
398 `END;`  
399 `ELSE`  
400 `NEW( p ); p.f := f;`  
401 `res := p`  
402 `END;`  
403 `RETURN res`  
404 `END NewPrimitive;`

405  
406 `PROCEDURE ( p: Primitive ) Y ( x: REAL ): REAL;`  
407 `CONST n = 10000; VAR N: INTEGER;`  
408 `BEGIN`  
409 `N := SHORT( ENTIER( N * x ) );`  
410 `RETURN Integral( p.f, 0, x, N )`  
411 `END Y;`

412  
413 **MODULE Kurs2006FunctionsEx1;** *полный текст*

414  
415  
416 0) Получаем возможность делать *несколько* производных/первообразных.

417 `d1 := Derivative( f );`  
418 `d2 := Derivative( d1 );`

419 1) Там, где можно точно взять интеграл или производную, они берутся точно. В  
420 остальных случаях -- приближенные формулы.

421 2) Производные и первообразные "знают" друг про друга.

422  
423 Дальнейшие возможности развития:

424 -- операции с функциями (сумма полиномов -- полином, и т.д.)

```

425 -- интерполяции (Ньютон, лин./кубич. сплайны, ...)
426 -- обращение функций (монотонность)
427 -- композиция функций
428 (напр., графики в логарифмич. масштабе)
429 ...
430
431 Недостатки
432 Главный недостаток: слишком конкретная информация зашита в текст -- в
433 процедурах NewDerivative, NewPrimitive -- в операторах WITH; в конкретных
434 процедурах численного вычисления производных и интегралов.
435 Хотелось бы большей гибкости:
436   — узнавать, когда у функции производная вычислилась точно, а когда нет;
437   — узнавать о свойствах регулярности функции, чтобы выбирать соотв.
438 алгоритмы;
439   — иметь возможность выбирать алгоритм вычисления производной и т.п.
440 NB Хотелось бы независимо развивать "базовые" модули с алгоритмами, и
441 "клиенские" модули с конкретными функциями.
442
443 Применим схему, уже апробированную в случае с Views:
444
445         Function <--> View
446             Y <--> Restore
447         HandleMsg <--> HandlePropMsg
448
449 MODULE Kurs2006Functions;
450
451     TYPE
452         Function* = POINTER TO ABSTRACT RECORD END;
453         Message* = ABSTRACT RECORD END;
454
455     PROCEDURE ( f: Function ) Y* ( x: REAL ): REAL, NEW, ABSTRACT;
456     PROCEDURE ( f: Function ) HandleMsg* ( VAR msg: Message ), NEW, EMPTY;
457
458 END Kurs2006Functions.
459
460 В случае с View было два типа мессиджей -- в модулях Properties, Controllers --
461 и два соотв. метода: HandlePropMsg и HandleCtrlMsg. Здесь (пока) один.
462
463 Эта схема (впервые, to the best of my knowledge) систематически применена в
464 развивающейся библиотеке Numath ((с) Ф.В.Ткачев 2005-2007) для обработки
465 данных в рамках эксперимента Троицк-V-mass по поиску массы нейтрино в  $\beta$ -
466 распаде трития (ИЯИ РАН, рук. акад. В.М.Лобашев).
467
468 Производные
469 Сосредоточимся на производных. Производные нужны во многих случаях.
470 Обычно (на фортране, паскале, ц, ц++ .. всюду, где нет автоматич. упр.
471 памятью) включать в алгоритмы производные — головная боль с т. зр.
472 программной организации.
473 Известен даже прожект, когда была сделана программа, которая брала
474 исходники и производила снова исходники для вычисления соотв. производных
475 (!?):
476

```

```

477 ".. The unique capability I was referring to is automatic differentiation (http://www-fp.mcs.anl.gov/autodiff/). Derivatives are notoriously difficult to compute numerically with accuracy, but a method has been found to avoid numerical approximations by operating directly on source code. It works only with ancient FORTRAN 77 and C .."
478
479
480
481
482
483 Поэтому построим модуль для "умных" функций, умеющих себя
484 дифференцировать.
485
486 MODULE Kurs2006Functions;
487
488     TYPE
489         Function* = POINTER TO ABSTRACT RECORD END;
490
491         SimpleDx = POINTER TO RECORD ( Function ) (* не экспортируем *)
492             dx: REAL;
493             f: Function
494         END;
495
496         Message* = ABSTRACT RECORD END;
497
498         ExactDxMsg* = RECORD ( Message )
499             d*: Function
500         END;
501
502     PROCEDURE ( f: Function ) Y* ( x: REAL ): REAL, NEW, ABSTRACT;
503     PROCEDURE ( f: Function ) HandleMsg* ( VAR msg: Message ), NEW, EMPTY;
504
505
506     PROCEDURE Deriv0 ( f: Function; x, dx: REAL ): REAL;
507         VAR res: REAL;
508     BEGIN
509         RETURN ( f.Y( x + dx/2 ) - f.Y( x - dx/2 ) ) / dx
510         (* надо бы учитывать области определения -- для этого отдельный запрос *)
511     END Deriv0;
512
513
514     PROCEDURE ( d: SimpleDx ) Y ( x: REAL ): REAL;
515     BEGIN
516         RETURN Deriv0( d.f, x, d.dx )
517         (* надо бы лучше алгоритмы -- Рунге, Эйткен и т.п. *)
518     END Y;
519
520     PROCEDURE NewSimpleDx* ( f: Function; dx: REAL ): Function;
521         VAR res: SimpleDx;
522     BEGIN
523         ASSERT( f # NIL, 20 ); ASSERT( dx # 0, 21 );
524         NEW( res ); res.f := f; res.dx := dx; RETURN res
525     END NewSimpleDx;
526
527
528     PROCEDURE Derivative* ( f: Function; dx: REAL ): Function;
529         VAR msg: ExactDxMsg; res: Function;

```

```

530 BEGIN
531   ASSERT( f # NIL, 20 );
532   msg.d := NIL; (* подготовили мессидж *)
533   f.HandleMsg( msg ); (* опрашиваем (=poll) функцию f *)
534   IF msg.d # NIL THEN (* f ответила "да, знаю свою точную производную"
535   *)
536     res := msg.d
537   ELSE (* f не среагировала -- действуем по своему усмотрению *)
538     res := NewSimpleDx( f, dx );
539   END;
540   RETURN res
541 END Derivative;
542
543 END Kurs2006Functions.
544
545 NB Все возможности ("компоненты") доступны по отдельности.
546 Единственная функция, где все одновременно -- это Derivative, по замыслу
547 предлагаемая как простой и разумный вариант по умолчанию.
548 Но у клиента остается возможность соорудить свой вариант.
549
550 Клиенты не видят, что за конкретный тип выдает Derivative, — только что это
551 какой-то потомок типа Function, т.е. знает, как себя вычислять: Y(x).
552
553 NB Нехорошо привязывать процедуру Derivative к Function: d := f.Derivative();
554 это было бы "намертво".
555
556 NB Мы только "расширили" интерфейс (добавили новые процедуры и методы
557 после тех, что уже были в модуле), никак не модифицируя старый интерфейс —
558 значит, все клиенты, скомпилированные ранее под старый интерфейс (напр.,
559 вычисление интеграла, нуля и т.п.) будут работать — их перекомпилировать
560 не надо.
561 Это называется независимая эволюция программных компонент, из
562 которых составляется программа -- нам постепенно раскрывается смысл слов
563 component-oriented programming.
564
565 Можно придумать целые независимые модули-словари вопросов, на которые
566 могли бы отвечать функции. Но нелегко придумать удобный (универсальный,
567 простой в использовании) набор сообщений для опроса функций -- удобство
568 выясняется только в процессе "обкатки" в реальных задачах.
569
570 NB В каждой задаче мы можем ограничиться только теми свойствами, которые
571 там конкретно играют роль -- и игнорировать остальные.
572
573 Пример конкретного потомка, знающего свои точные производные:
574
575 MODULE Kurs2006Polynomials;
576   IMPORT F := Kurs2006Functions;
577
578   TYPE
579     Polynomial = POINTER TO RECORD ( F.Function )
580     (* без нужды не экспортируем *)
581     c: POINTER TO ARRAY OF REAL (* коэффициенты *)
582   END;

```

```

583
584 PROCEDURE ( p: Polynomial ) Y ( x: REAL ): REAL; (* схема Горнера *)
585   VAR res: REAL; i: INTEGER;
586 BEGIN
587   res := 0;
588   FOR i := LEN( p.c ) - 1 TO 0 BY -1 DO
589     res := p.c[ i ] + res * x
590   END;
591   RETURN res
592 END Y;
593
594 PROCEDURE ( p: Polynomial ) HandleMsg ( VAR msg: F.Message );
595   VAR d: Polynomial; i: INTEGER;
596 BEGIN
597   WITH msg: F.ExactDxMsg DO
598     NEW( d );
599     NEW( d.c, LEN( p.c ) - 1 );
600     FOR i := 0 TO LEN( d.c ) - 1 DO
601       d.c[ i ] := ( i + 1 ) * p.c[ i + 1 ]
602     END;
603     msg.d := d
604   ELSE
605     END;
606 END HandleMsg;
607
608 PROCEDURE NewPolynomial* ( IN c: ARRAY OF REAL ): F.Function;
609   (* фабричная функция *)
610   (* раз не сделали видимым тип, то и незачем указывать его в сигнатуре *)
611   VAR p: Polynomial; i: INTEGER;
612 BEGIN
613   NEW( p ); NEW( p.c, LEN( c ) );
614   FOR i := 0 TO LEN( c ) - 1 DO
615     p.c[ i ] := c[ i ]
616   END;
617   RETURN p
618 END NewPolynomial;
619
620 END Kurs2006Polynomials.
621
622 NB Алгоритмы теперь обычно следует писать, используя d := F.Derivative( f );
623
624 NB Для любых Functions можно написать
625   d2 := F.Derivative( F.Derivative( f ) ) — и получим функцию, которая
626 является второй производной (хотя точность вычисления может быть
627 плоховата).
628 Но для полиномов автоматически будет точная формула!
629
630 NB При разработке нового конкретного класса функций сначала для простоты
631 могли бы проигнорировать точные производные -- будет использоваться
632 "затычка", предлагаемая в модуле ***Functions по умолчанию.
633 Ведь слишком рано добавлять детали нехорошо (принцип top-
634 down/пошаговая детализация).
635 По мере необходимости можно добавлять информацию о точных производных.

```

```

636
637 ***NB He вводим
638   PROCEDURE ( f: Function ) ExactDx ( ), NEW, ABSTRACT;
639   т.к. пришлось бы обязательно реализовывать ... морока.
640
641 NB Что делать, чтобы разрешить клиенту использовать свою реализацию
642 "производной по умолчанию"?
643 Ввести в Functions еще один уровень "косвенности" (indirection): подобно dir:
644 Directory в Files, и процедуру NewSimpleDx привязать к этому dir + дать
645 процедуру SetDir, позволяющую "подменять" dir "на лету".
646
647 Два типа клиентов:
648   которые просто пользуются готовым;
649   которые достраивают что-то свое.
650
651 Дле первых важна простота использования: df := F.Derivative( f );
652 Для последних важно, чтобы усложнения можно было вводить постепенно.
653
654 Описанным способом удовлетворим всех.
655
656 Аналогично: тригонометр. функции.
657
658   TYPE Trig = POINTER TO RECORD ( F.Function ) a, b: REAL END;
659
660   PROCEDURE ( t: Trig ) Y ( x: REAL ): REAL;
661   BEGIN RETURN a * Math.Sin( x ) + b * Math.Cos( x )
662   END Eval;
663
664   PROCEDURE ( t: Trig ) HandleMsg ( msg: F.Message );
665   VAR d: Trig;
666   BEGIN
667     WITH msg: ExactDerivMsg DO
668       NEW( d ); d.a := -t.b; d.b := t.a; msg.d := d
669     ELSE
670       END
671   END Domain;
672
673 Можно аналогично производным определить
674 — обращение функций;
675 — композицию функций; (то и другое для е.г. организации лог. шкалы на
676 графиках)
677 — арифметику функций;
678 причем связать все это с производными ...
679 Упр Обдумать.
680
681 Кроме производных можно вычислять интерполирующие полиномы, фурье-
682 аппроксимации и т.п.
683
684 NB Часто полезно узнавать, является ли функция: константой (1 или 0),
685 тождественным отображением (Y(x)=x) и т.п. Тождественным нулем может
686 оказаться как полином, так и тригонометрич. функция. Громоздить иерархию
687 наследования для этой цели трудно (даже может быть невозможно, если
688 хочется сохранить полную независимость реализаций триг. функций и

```

689 полиномов -- а сохранить независимость полезно: DIVIDE ET IMPERA!). Здесь  
690 эта схема -- "**шина сообщений/Oberon software bus**" -- срабатывает  
691 **идеально**. Как и с выюшками.

692  
693 NB Полезно иметь в виду возможность сделать Function наследником  
694 Stores.Store -- тогда можно сохранять в файл (нужно, конечно, определить  
695 Internalize/Externalize). Например, если функция есть результат сложных  
696 вычислений/интерполяций/фитирования -- или просто если она сложная и при  
697 обращениях к Y(x) накапливает информацию, чтобы при последующих вызовах  
698 по возможности использовать интерполяцию.

## Еще раз правила дизайна

703 Если предполагается, что клиент будет "настраивать" каркас, определяя свои  
704 типы-потомки, то интерфейс должен состоять только из ABSTRACT-типов и  
705 ABSTRACT|EMPTY-методов, а также конкретных процедур.  
706 В менее ответственных ситуациях можно также LIMITED, и совсем в простых —  
707 без атрибутов (FINAL).

708  
709 NB EXTENSIBLE (за редким исключением) — только внутри модуля, для  
710 неэкспортируемых типов.  
711 Главным образом, EXTENSIBLE -- анестетик при переносе программ из старого  
712 Оберона и др. языков.

713  
714 Для этих правил есть основания (*fragile base class problems*): при  
715 беспорядочном наследовании может стать невозможным дальнейшая  
716 независимая эволюция клиентов и базового типа.

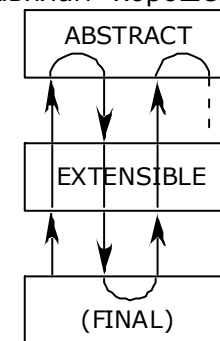
717  
718 Достаточно следовать сформулированным правилам "хорошего  
719 тона".

720  
721 Еще терминология: когда из модуля Functions во время  
722 выполнения реально вызывается процедура, конкретно  
723 определенная в модуле-клиенте, который сам вызвал  
724 что-то из Function, то это называется **callback** или  
725 **upcall**.

726 Цепочка вызовов процедур может приобрести  
727 головокрумную запутанность (т.наз. **yo-yo pattern**):

728  
729 Снова: **KEEP IT SIMPLE, STUPID**

730



731 **Кривые Безье**

732 Очень полезный инструмент для рисования гладких кривых, не слишком  
733 сложный.

734 Требуется вычисления производных — и для традиц. программирования  
735 потихоньку начинается головная боль ...

736  
737 Соединим две точки

$(X_0, Y_0), (X_1, Y_1)$

738 следующей кривой:

$$739 \quad x(t) = x_0 + x_1 t + x_2 t^2 + x_3 t^3, \\ 740 \quad y(t) = y_0 + y_1 t + y_2 t^2 + y_3 t^3, \quad 0 \leq t \leq 1.$$

742 Имеется 8 коэффициентов. Нужно 8 условий. Вот они:

$$x(0) = X_0, \quad x(1) = X_1; \quad x'(0) = X'_0, \quad x'(1) = X'_1;$$

$$743 \quad y(0) = Y_0, \quad y(1) = Y_1; \quad y'(0) = Y'_0, \quad y'(1) = Y'_1.$$

745 Соответствующие линейные уравнения для коэффициентов  $x$  таковы:

$$x(0) = X_0: \quad x_0 = X_0;$$

$$x(1) = X_1: \quad x_1 + x_2 + x_3 = X_1 - X_0;$$

$$x'(0) = X'_0: \quad x_1 = X'_0;$$

$$746 \quad x'(1) = X'_1: \quad x_1 + 2x_2 + 3x_3 = X'_1.$$

747 Уравнения для  $y$  аналогичны.

748

749 Находим:

$$x_0 = X_0, \quad x_1 = X'_0,$$

$$x_2 = X_1 - X_0 - x_1 - x_3;$$

$$750 \quad x_3 = X'_1 + x_1 - 2(X_1 - X_0).$$

751 Выражения для  $y$  аналогичны.

752

753 **Пример**

754

$$(X_0, Y_0) = (0, 0); \quad (X_1, Y_1) = (1, 0);$$

$$(X'_0, Y'_0) = (0, 1); \quad (X'_1, Y'_1) = (0, -1).$$

755 Тогда

$$756 \quad x(t) = t^2(3 - 2t), \quad y(t) = t - t^2.$$

757 При этом

$$758 \quad y \sim \frac{1}{\sqrt{3}}\sqrt{x}, \quad x \rightarrow 0 \quad x(\frac{1}{2}) = \frac{1}{2}, \quad y(\frac{1}{2}) = \frac{1}{4}.$$

760

761 **Рисование кривой**

762

763 Пусть  $c(t)$  - кривая на плоскости (два вещественных значения для каждого  $t$ ), и  
764 мы хотим представить ее одним сегментом Безье между  $t_0$  и  $t_1$ . Тогда 4 точки  
765 Безье таковы:

$$P_0 = c(t_0); \quad P_1 = P_0 + c'(t_0) \frac{(t_1 - t_0)}{3};$$

$$P_3 = c(t_1); \quad P_2 = P_3 - c'(t_1) \frac{(t_1 - t_0)}{3}.$$

766

767 Здесь

$$c'(t) = \frac{dc(t)}{dt}$$

768

769

770 **Рисование функции**

771

772 Если функция  $f(x)$ , которую нужно изобразить, является гладкой, то достаточно  
773 отождествить  $x=t$ ,  $y=f(t)$ . Тогда

$$c(t) = \begin{pmatrix} x \\ f(x) \end{pmatrix} \quad c'(t) = \begin{pmatrix} 1 \\ f'(x) \end{pmatrix}$$

774

775 и четыре точки Безье таковы:

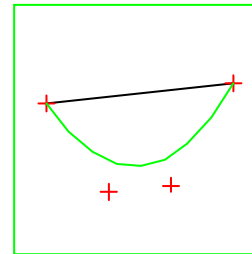
$$P_0 = \begin{pmatrix} x_0 \\ f(x_0) \end{pmatrix}; \quad P_1 = P_0 + \begin{pmatrix} 1 \\ f'(x_0) \end{pmatrix} \frac{(t_1 - t_0)}{3};$$

$$P_3 = \begin{pmatrix} x_1 \\ f(x_1) \end{pmatrix}; \quad P_2 = P_3 - \begin{pmatrix} 1 \\ f'(x_1) \end{pmatrix} \frac{(t_1 - t_0)}{3}.$$

776

777

778 Иллюстрация -- как по четырем точкам строится сегмент:



779

780

781

782 **MODULE Kurs2006Functions;**

783 **IMPORT** Stores, Math;

784 **CONST** minVersion = 7; maxVersion = 7; (\* versions 0/2002, 1/2004 etc. are  
785 obsolete \*)

786

787

788 (\* Directory not implemented in this version \*)

789 **Function\*** = POINTER TO ABSTRACT RECORD ( Stores.Store ) **END**; (\*  
790 treated as immutable \*)

791

792 **Message\*** = ABSTRACT RECORD **END**;

793

794 **ExactDx\*** = RECORD ( Message )

795 **d\***: Function

796 **END**;

```

797
798  (*****)
799  (* Конкретные классы функций, известные модулю *)
800
801  Linear = POINTER TO LIMITED RECORD ( Function )
802    a, b: REAL (*  $a*x + b$  *)
803  END;
804
805  Sin = POINTER TO LIMITED RECORD ( Function )
806    a: REAL (*  $\sin(x + a)$ ; perhaps in future:  $a*\sin(b*x + c)$  *)
807  END;
808  (* пока недостаточно опыта, чтобы решить, как удобно сделать;
809    in future:  $\text{ArcSin}$  *)
810
811  Polynomial = POINTER TO LIMITED RECORD ( Function )
812    c: POINTER TO ARRAY OF REAL (*  $\sum_i c[i] * x^i$  *)
813  END;
814
815  (*****)
816
817  SimpleDx = POINTER TO LIMITED RECORD ( Function ) (* не
818    экспортируем *)
819    f: Function;
820    dx: REAL;
821    (* d: Segment *)
822  END;
823
824  (* для графиков с нелинейными координатами: *)
825  ExactComposition* = RECORD ( Message )
826    g*, h*: Function;
827  END;
828
829  Comp = POINTER TO LIMITED RECORD ( Function )
830    f, g: Function
831  END;
832
833  VAR id-, sin-, cos -: Function;
834  VAR piOver2: REAL;
835
836
837  PROCEDURE ( f: Function ) Y* ( x: REAL ): REAL, NEW, ABSTRACT;
838  PROCEDURE ( f: Function ) HandleMsg* ( VAR msg: Message ), NEW, EMPTY;
839
840
841  (* Фасады [ср. с Views] *)
842
843  PROCEDURE ReadFunction* ( VAR rd: Stores.Reader; OUT f: Function );
844    VAR s0: Stores.Store;
845  BEGIN rd.ReadStore( s0 ); f := s0( Function )
846  END ReadFunction;
847
848  PROCEDURE WriteFunction* ( VAR wr: Stores.Writer; f: Function );
849  BEGIN ASSERT( f # NIL, 20 ); wr.WriteStore( f )

```

```

850  END WriteFunction;
851
852
853  (** Linear *****)
854
855  (* нужна "фабричная функция" [еще гибче было бы с ф. объектом..] *)
856  PROCEDURE NewLinear* ( a, b: REAL ): Function;
857    VAR l: Linear;
858  BEGIN
859    ASSERT( ABS( a ) # INF, 20 ); ASSERT( ABS( b ) # INF, 21 );
860    NEW( l ); l.a := a; l.b := b; RETURN l
861  END NewLinear;
862
863  (* единственный обязательный метод: *)
864  PROCEDURE ( l: Linear ) Y ( x: REAL ): REAL;
865  BEGIN
866    ASSERT( ( x < INF ) & ( x > -INF ), 20 ); (* береженого Бог ... *)
867    RETURN l.a * x + l.b
868  END Y;
869
870  (* Стандартное занудство, необязательное в простых задачах: *)
871
872  PROCEDURE ( l: Linear ) Externalize ( VAR wr: Stores.Writer );
873  BEGIN
874    wr.WriteVersion( maxVersion );
875    wr.WriteReal( l.a ); wr.WriteReal( l.b ) (****)
876  END Externalize;
877
878  PROCEDURE ( l: Linear ) Internalize ( VAR rd: Stores.Reader );
879    VAR version: INTEGER;
880  BEGIN
881    rd.ReadVersion( minVersion, maxVersion, version );
882    IF ~rd.cancelled THEN (* version в допустимом интервале *)
883      rd.ReadReal( l.a ); rd.ReadReal( l.b ) (****)
884    END
885  END Internalize;
886
887  PROCEDURE ( l: Linear ) CopyFrom ( source: Stores.Store );
888  BEGIN
889    WITH source: Linear DO
890      l.a := source.a; l.b := source.b (****)
891    END;
892  END CopyFrom;
893
894  (* необязательный, но полезный метод: *)
895  PROCEDURE ( l: Linear ) HandleMsg ( VAR msg: Message );
896    VAR d: Linear; i: Linear; g: Function;
897  BEGIN
898    WITH msg: ExactDx DO
899      NEW( d ); d.a := 0; d.b := l.a;
900      msg.d := d
901    | msg: ExactComposition DO

```

```

903     ASSERT( msg.g # NIL, 40 ); ASSERT( msg.h = NIL, 41 );
904     g := msg.g;
905     WITH g: Linear DO
906         msg.h := NewLinear( l.a * g.a, l.a * g.b + l.b )
907     ELSE
908         ASSERT( msg.h = NIL, 40 );
909     END
910 ELSE
911     END;
912 END HandleMsg;
913
914
915 (** Sin *****)
916
917 PROCEDURE NewSin* ( a: REAL ): Function;
918     VAR s: Sin;
919 BEGIN
920     NEW( s ); s.a := a; RETURN s
921 END NewSin;
922
923 PROCEDURE ( s: Sin ) Y ( x: REAL ): REAL;
924 BEGIN RETURN Math.Sin( x + s.a )
925 END Y;
926
927 PROCEDURE ( s: Sin ) Externalize ( VAR wr: Stores.Writer );
928 BEGIN
929     wr.WriteVersion( maxVersion );
930     wr.WriteReal( s.a )
931 END Externalize;
932
933 PROCEDURE ( s: Sin ) Internalize ( VAR rd: Stores.Reader );
934     VAR version: INTEGER;
935 BEGIN
936     rd.ReadVersion( minVersion, maxVersion, version );
937     IF ~rd.cancelled THEN
938         rd.ReadReal( s.a )
939     END
940 END Internalize;
941
942 PROCEDURE ( s: Sin ) CopyFrom ( source: Stores.Store );
943 BEGIN
944     WITH source: Sin DO
945         s.a := source.a
946     END;
947 END CopyFrom;
948
949 PROCEDURE ( s: Sin ) HandleMsg ( VAR msg: Message );
950     VAR d: Sin;
951 BEGIN
952     WITH msg: ExactDx DO
953         ASSERT( msg.d = NIL, 20 );
954         NEW( d ); d.a := s.a + piOver2;
955         msg.d := d

```

```

956     ELSE
957     END
958 END HandleMsg;
959
960
961 (** Polynomial *****)
962
963 PROCEDURE NewPolynomial* ( IN a: ARRAY OF REAL; n: INTEGER ):
964 Function;
965     VAR p: Polynomial; i: INTEGER;
966 BEGIN
967     ASSERT( n <= LEN( a ), 20 );
968     NEW( p ); NEW( p.c, n );
969     FOR i := 0 TO n - 1 DO p.c[ i ] := a[ i ] END;
970     RETURN p
971 END NewPolynomial;
972
973 PROCEDURE ( p: Polynomial ) Degree (): INTEGER, NEW; (* для
974 самодokumentированности — чтобы работать только в терминах Degree, а не
975 LEN( p.c ) *)
976 BEGIN
977     RETURN LEN( p.c ) - 1
978 END Degree;
979
980 PROCEDURE ( p: Polynomial ) Y ( x: REAL ): REAL; (* Gerner *)
981     VAR res: REAL; i: INTEGER;
982 BEGIN
983     res := 0;
984     FOR i := p.Degree() TO 0 BY -1 DO
985         res := p.c[ i ] + res * x
986     END;
987     RETURN res
988 END Y;
989
990 PROCEDURE ( p: Polynomial ) Externalize ( VAR wr: Stores.Writer );
991     VAR i, n: INTEGER;
992 BEGIN
993     wr.WriteVersion( maxVersion );
994     n := p.Degree();
995     wr.WriteInt( n );
996     FOR i := 0 TO n DO wr.WriteReal( p.c[ i ] ) END
997 END Externalize;
998
999 PROCEDURE ( p: Polynomial ) Internalize ( VAR rd: Stores.Reader );
1000     VAR i, n, version: INTEGER;
1001 BEGIN
1002     rd.ReadVersion( minVersion, maxVersion, version );
1003     IF ~rd.cancelled THEN
1004         rd.ReadInt( n ); NEW( p.c, n + 1 );
1005         FOR i := 0 TO n DO rd.ReadReal( p.c[ i ] ) END
1006     END
1007 END Internalize;
1008

```

```

1009 PROCEDURE ( p: Polynomial ) CopyFrom ( source: Stores.Store );
1010 BEGIN
1011     WITH source: Polynomial DO
1012         p.c := source.c; (* NB *)
1013     END;
1014 END CopyFrom;
1015
1016 PROCEDURE ( p: Polynomial ) HandleMsg ( VAR msg: Message );
1017 VAR d: Polynomial; lo, hi: REAL; i, n: INTEGER;
1018 BEGIN
1019     WITH msg: ExactDx DO
1020         ASSERT( msg.d = NIL, 20 );
1021         n := p.Degree();
1022         NEW( d ); NEW( d.c, n );
1023         FOR i := 0 TO n - 1 DO
1024             d.c[ i ] := ( i + 1 ) * p.c[ i + 1 ]
1025         END;
1026         msg.d := d
1027     | msg: ExactComposition DO
1028         ASSERT( msg.g # NIL, 40 ); ASSERT( msg.h = NIL, 41 );
1029     ELSE
1030     END
1031 END HandleMsg;
1032
1033 (*****
1034 (* Попробуем дать реализацию с разумным ratio надежность / простота. *)
1035
1036 PROCEDURE NewSimpleDx* ( f: Function; dx: REAL ): Function; (* Трудно
1037     выбрать dx: пусть сам юзер контролирует. *)
1038 VAR res: SimpleDx;
1039 BEGIN
1040     ASSERT( f # NIL, 20 ); ASSERT( dx # 0, 21 );
1041     NEW( res ); res.f := f; res.dx := dx; RETURN res
1042 END NewSimpleDx;
1043
1044 PROCEDURE Deriv0 ( f: Function; x, dx: REAL ): REAL;
1045 VAR res: REAL;
1046 BEGIN
1047     RETURN ( f.Y( x + dx/2 ) - f.Y( x - dx/2 ) ) / dx
1048     (* надо бы учитывать области определения *)
1049 END Deriv0;
1050
1051 PROCEDURE ( d: SimpleDx ) Y ( x: REAL ): REAL;
1052 BEGIN
1053     RETURN Deriv0( d.f, x, d.dx )
1054     (* надо бы получить алгоритмы -- Рунге, Эйткен и т.п. *)
1055 END Y;
1056
1057 PROCEDURE ( d: SimpleDx ) Externalize ( VAR wr: Stores.Writer );
1058 BEGIN
1059     wr.WriteVersion( maxVersion );
1060     wr.WriteReal( d.dx );
1061     WriteFunction( wr, d.f );
1062     END Externalize;

```

```

1062 PROCEDURE ( d: SimpleDx ) Internalize ( VAR rd: Stores.Reader );
1063 VAR version: INTEGER;
1064 BEGIN
1065     rd.ReadVersion( minVersion, maxVersion, version );
1066     IF ~rd.cancelled THEN
1067         rd.ReadReal( d.dx );
1068         ReadFunction( rd, d.f );
1069     END
1070 END Internalize;
1071
1072 PROCEDURE ( d: SimpleDx ) CopyFrom ( source: Stores.Store );
1073 BEGIN
1074     WITH source: SimpleDx DO
1075         d.dx := source.dx;
1076         d.f := source.f; (* NB *)
1077     END;
1078 END CopyFrom;
1079
1080 PROCEDURE NewDerivative* ( f: Function; dx: REAL ): Function;
1081 VAR msg: ExactDx; res: Function;
1082 BEGIN ASSERT( f # NIL, 20 );
1083     msg.d := NIL; (* для надежности *)
1084     f.HandleMsg( msg );
1085     IF msg.d # NIL THEN
1086         res := msg.d
1087     ELSE
1088         res := NewSimpleDx( f, dx );
1089         (* могли бы тут определять подходящий dx *)
1090     END;
1091     RETURN res
1092 END NewDerivative;
1093
1094 PROCEDURE NewSimpleComposition* ( f, g: Function ): Function;
1095 VAR res: Comp;
1096 BEGIN ASSERT( f # NIL, 20 ); ASSERT( g # NIL, 21 );
1097     NEW( res ); res.f := f; res.g := g; RETURN res
1098 END NewSimpleComposition;
1099
1100 PROCEDURE ( c: Comp ) Y ( x: REAL ): REAL;
1101 BEGIN RETURN c.f.Y( c.g.Y( x ) )
1102 END Y;
1103
1104 PROCEDURE ( c: Comp ) HandleMsg ( VAR msg: Message );
1105 BEGIN
1106     WITH msg: ExactDx DO
1107         ASSERT( msg.d = NIL, 20 );
1108         (* если достаточно информации, то цепная формула -- Упр *)
1109     | msg: ExactComposition DO
1110         ASSERT( msg.g # NIL, 40 ); ASSERT( msg.h = NIL, 41 );
1111         (* например, композиция двух Linear есть снова Linear -- Упр *)
1112     ELSE
1113     END;
1114 END HandleMsg;

```

```

1115
1116 PROCEDURE ( c: Comp ) Externalize ( VAR wr: Stores.Writer );
1117 BEGIN
1118     wr.WriteVersion( maxVersion );
1119     WriteFunction( wr, c.f );
1120     WriteFunction( wr, c.g );
1121 END Externalize;
1122
1123 PROCEDURE ( c: Comp ) Internalize ( VAR rd: Stores.Reader );
1124     VAR version: INTEGER;
1125 BEGIN
1126     rd.ReadVersion( minVersion, maxVersion, version );
1127     IF ~rd.cancelled THEN
1128         ReadFunction( rd, c.f );
1129         ReadFunction( rd, c.g );
1130     END
1131 END Internalize;
1132
1133 PROCEDURE ( c: Comp ) CopyFrom ( source: Stores.Store );
1134 BEGIN
1135     WITH source: Comp DO
1136         c.f := source.f; c.g := source.g (* NB *)
1137     END;
1138 END CopyFrom;
1139
1140 PROCEDURE NewComposition* ( f, g: Function ): Function; (* фасад *)
1141     VAR res: Function;
1142 BEGIN
1143     ASSERT( f # NIL, 20 ); ASSERT( g # NIL, 21 );
1144     res := NewSimpleComposition( f, g ); (* первая тупая реализация *)
1145     RETURN res
1146 END NewComposition;
1147
1148 PROCEDURE Init;
1149 BEGIN
1150     piOver2 := Math.Pi() / 2;
1151     id := NewLinear( 1, 0 );
1152     sin := NewSin( 0 );
1153     cos := NewSin( piOver2 );
1154 END Init;
1155
1156 BEGIN Init
1157 END Kurs2006Functions.
1158
1159

```

```

1160 MODULE Kurs2006Bezier;
1161     IMPORT Log := StdLog, Views, Ports, StdCmds, Stores, Properties,
1162         F := Kurs2006Functions, Math, In;
1163
1164     CONST mm = Ports.mm; defH = 35 * mm; defW = defH; margin = 1 * mm;
1165         minsize = 2 * mm + 2 * margin;
1166
1167     TYPE
1168         FramePoint = Ports.Point;
1169         FramePoints = POINTER TO ARRAY OF FramePoint;
1170         Point = RECORD x, y: REAL END;
1171         Points = POINTER TO ARRAY OF Point;
1172
1173         View* = POINTER TO RECORD ( Views.View )
1174             fx, fy: F.Function; (* рисуемая кривая *)
1175             p0, p1: Point; (* границы картинки в естеств. коорд. *)
1176             t0, t1: REAL; (* концы рисуемой части кривой *)
1177             n: INTEGER
1178         END;
1179
1180     PROCEDURE ( v: View ) Externalize- ( VAR wr: Stores.Writer );
1181     BEGIN
1182         F.WriteFunction( wr, v.fx ); F.WriteFunction( wr, v.fy );
1183         wr.WriteReal( v.p0.x ); wr.WriteReal( v.p0.y );
1184         wr.WriteReal( v.p1.x ); wr.WriteReal( v.p1.y );
1185         wr.WriteReal( v.t0 ); wr.WriteReal( v.t1 );
1186         wr.WriteInt( v.n )
1187     END Externalize;
1188
1189     PROCEDURE ( v: View ) Internalize- ( VAR rd: Stores.Reader );
1190     BEGIN
1191         F.ReadFunction( rd, v.fx ); F.ReadFunction( rd, v.fy );
1192         rd.ReadReal( v.p0.x ); rd.ReadReal( v.p0.y );
1193         rd.ReadReal( v.p1.x ); rd.ReadReal( v.p1.y );
1194         rd.ReadReal( v.t0 ); rd.ReadReal( v.t1 );
1195         rd.ReadInt( v.n )
1196     END Internalize;
1197
1198     PROCEDURE ( v: View ) CopyFromSimpleView- ( source: Views.View );
1199     BEGIN
1200         WITH source: View DO
1201             v.fx := source.fx; v.fy := source.fy; (* NB "функциональный стиль"
1202             *)
1203             v.p0 := source.p0; v.p1 := source.p1;
1204             v.t0 := source.t0; v.t1 := source.t1;
1205             v.n := source.n
1206         END;
1207     END CopyFromSimpleView;
1208

```

```

1209 PROCEDURE ( v: View ) HandlePropMsg- ( VAR p: Properties.Message );
1210 BEGIN
1211     WITH p: Properties.SizePref DO
1212         IF p.w = Views.undefined THEN (* w и h всегда undefined
1213 одновременно *)
1214             p.w := defW; p.h := defH
1215         ELSE (* будем менять размер картинки независимо от размера окна
1216 (Ctrl+space) *)
1217             p.w := MAX( p.w, minsize ); p.h := MAX( p.h, minsize )
1218         END;
1219     | p: Properties.ResizePref DO (* размер картинки привязан к размеру
1220 окна *)
1221         p.horFitToWin := TRUE; p.verFitToWin := TRUE
1222     ELSE
1223         END;
1224     END HandlePropMsg;
1225
1226 PROCEDURE Translate ( p, p0, p1: Point; xscale, yscale: INTEGER; OUT fp:
1227 FramePoint );
1228 BEGIN
1229     fp.x := margin + SHORT( ENTIER( ( p.x - p0.x ) / ( p1.x - p0.x ) * xscale ));
1230     fp.y := margin + SHORT( ENTIER( ( p1.y - p.y ) / ( p1.y - p0.y ) * yscale ))
1231 END Translate;
1232
1233 PROCEDURE TranslatePoints ( p: Points; p0, p1: Point; xscale, yscale:
1234 INTEGER ): FramePoints;
1235 VAR i: INTEGER; fp: FramePoints;
1236 BEGIN
1237     NEW( fp, LEN( p ) );
1238     FOR i := 0 TO LEN( p ) - 1 DO Translate( p[ i ], p0, p1, xscale, yscale,
1239 fp[ i ] ) END;
1240     RETURN fp
1241 END TranslatePoints;
1242
1243 PROCEDURE ( VAR p: Point ) Set ( fx, fy: F.Function; t: REAL ), NEW;
1244 BEGIN p.x := fx.Y( t ); p.y := fy.Y( t );
1245 END Set;
1246
1247 PROCEDURE BezierPoints ( fx, fy: F.Function; IN t: ARRAY OF REAL ): Points;
1248 VAR b: Points; i, nBezier, n, s: INTEGER; dfx, dfy: F.Function;
1249 d, p0, p1, p2, p3: Point; (* 4 точки Безье для одного сегмента *)
1250 delta, dx, t0, t1: REAL;
1251 BEGIN
1252     n := LEN( t );
1253     dx := ( t[ n - 1 ] - t[ 0 ] ) * 1.0E-5;
1254     dfx := F.NewSimpleDx( fx, dx );
1255     dfy := F.NewSimpleDx( fy, dx );
1256     nBezier := ( n - 1 ) * 3 + 1; NEW( b, nBezier );
1257
1258     (* цикл по сегментам *)
1259     b[ 0 ].Set( fx, fy, t[ 0 ] );
1260     t1 := t[ 0 ];
1261     FOR s := 1 TO n - 1 DO

```

```

1262         i := s * 3;
1263         t0 := t1; p0 := b[ i - 3 ];
1264         t1 := t[ s ]; p3.Set( fx, fy, t1 );
1265
1266         delta := ( t1 - t0 )/3;
1267         p1.x := p0.x + delta * dfx.Y( t0 );
1268         p1.y := p0.y + delta * dfy.Y( t0 );
1269
1270         p2.x := p3.x - delta * dfx.Y( t1 );
1271         p2.y := p3.y - delta * dfy.Y( t1 );
1272
1273         b[ i - 2 ] := p1; b[ i - 1 ] := p2; b[ i ] := p3
1274     END;
1275     RETURN b
1276 END BezierPoints;
1277
1278 PROCEDURE DrawCross ( f: Views.Frame; fp: FramePoint; color: Ports.Color );
1279 CONST eps = mm;
1280 BEGIN
1281     f.DrawLine( fp.x - eps, fp.y, fp.x + eps, fp.y, 0, color );
1282     f.DrawLine( fp.x, fp.y - eps, fp.x, fp.y + eps, 0, color )
1283 END DrawCross;
1284
1285 PROCEDURE ( v: View ) Restore* ( f: Views.Frame; l, t, r, b: INTEGER );
1286 VAR tt: POINTER TO ARRAY OF REAL; p: Points; fp: FramePoints;
1287 i, w, h, xscale, yscale: INTEGER; step: REAL;
1288 BEGIN
1289     v.context.GetSize( w, h );
1290     xscale := w - 2 * margin; yscale := h - 2 * margin;
1291     IF ( xscale > 0 ) & ( yscale > 0 ) THEN
1292         f.DrawRect( margin, margin, w - margin, h - margin, 0, Ports.green );
1293
1294         (* ломаная *)
1295         NEW( tt, v.n + 1 );
1296         step := ( v.t1 - v.t0 ) / v.n;
1297         FOR i := 0 TO v.n DO
1298             tt[ i ] := v.t0 + i * step
1299         END;
1300
1301         NEW( p, v.n + 1 );
1302         FOR i := 0 TO v.n DO
1303             p[ i ].x := v.fx.Y( tt[ i ] ); p[ i ].y := v.fy.Y( tt[ i ] )
1304         END;
1305
1306         fp := TranslatePoints( p, v.p0, v.p1, xscale, yscale );
1307         f.DrawPath( fp, LEN( fp ), 0, Ports.black, Ports.openPoly );
1308
1309         p := BezierPoints( v.fx, v.fy, tt );
1310         fp := TranslatePoints( p, v.p0, v.p1, xscale, yscale );
1311
1312         (* рисуем точки Безье *)
1313         FOR i := 0 TO LEN( fp ) - 1 DO DrawCross( f, fp[ i ], Ports.red ) END;
1314         (* рисуем кривую Безье *)
1315         f.DrawPath( fp, LEN( fp ), 0, Ports.green, Ports.openBezier );
1316     END
1317 END Restore;

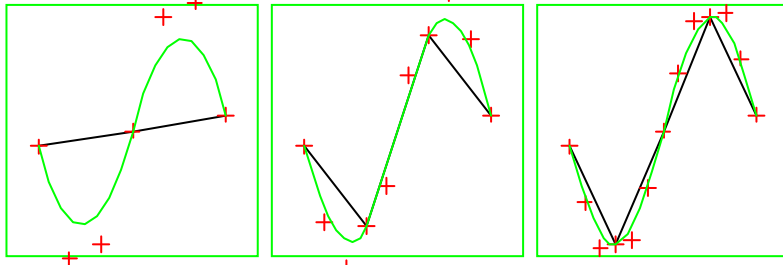
```

```

1311
1312 PROCEDURE New*( nSegm: INTEGER );
1313   VAR v: View; c: ARRAY 10 OF REAL; g0, g1: F.Function;
1314   BEGIN
1315     NEW( v );
1316
1317   (*   c[ 0 ] := 0.1; c[ 1 ] := 0; c[ 2 ] := -2;
1318       g0 := F.NewPolynomial( c, 3 );
1319
1320       c[ 0 ] := -1; c[ 1 ] := 1; c[ 2 ] := 2;
1321       g1 := F.NewPolynomial( c, 3 );
1322
1323       v.fx := F.NewComposition( g1, g0 ); *)
1324   v.fx := F.id;
1325
1326   (*   c[ 0 ] := 0.1; c[ 1 ] := -2; c[ 2 ] := 0; c[ 3 ] := 0.9;
1327       c[ 0 ] := -0.3; c[ 1 ] := 0.03; c[ 2 ] := 0.07; c[ 3 ] := -0.01;
1328       v.fy := F.NewPolynomial( c, 3 ); (* иллюстрация к математике *) *)
1329   v.fy := F.sin;
1330
1331   v.t0 := -3;
1332   v.t1 := +3;
1333   ASSERT( nSegm >= 1 ); v.n := nSegm;
1334   v.p0.x := -4; v.p1.x := +4; v.p0.y := -1.1; v.p1.y := 1.1;
1335   Views.OpenView( v )
1336   END New;
1337
1338 END Kurs2006Bezier.

```

1339 **!**"Kurs2006Bezier.New(2)"  
1340 **!**"Kurs2006Bezier.New(3)"  
1341 **!**"Kurs2006Bezier.New(4)"



1342  
1343  
1344  
1345  
1346  
1347  
1348

**NB** Сколько было бы еще мороки без автоматического управления памятью...

## 1349 Контекстное меню для нашей вьюшки

1350  
1351 Как активизировать меню только если в фокусе наша вьюшка?

```

1352
1353 PROCEDURE ( v: View ) HandleCtrlMsg ( f: Views.Frame; VAR msg:
1354   Controllers.Message; VAR focus: Views.View );
1355   VAR
1356   BEGIN
1357     WITH msg: Controllers.PollOpsMsg DO
1358       msg.type := "Kurs2006Bezier.View"
1359     ....
1360   END;
1361   END HandleCtrlMsg;
1362

```

1363 После этого где-нить в менюшных файлах можно вставить такое:

```

1364
1365 MENU "Мое мюню" ("Kurs2006Bezier.View")
1366   "Сделать нечто" "F7" "Модуль.Команда" ""
1367   ....

```

1368 Из команды тогда надо иметь доступ к вьюшке в фокусе: Всегда возможно:

```

1370   Controllers.FocusView (): Views.View;
1371

```

1372 после этого можно проверить тип и т.д.

1373  
1374  
1375  
1376  
1377  
1378