ТУБ №23:
Структура программы на языке Delphi

Создадим в Delphi новый проект консольного приложения. Среда автоматически генерирует каркас будущей программы, который может отличаться от версии к версии. Чтобы избежать обсуждения сложных концепций раньше времени, остановимся на более простом коде от Delphi 7-й версии:

```
program Project1;

{$APPTYPE CONSOLE}

uses
  SysUtils;

begin
  { TODO -oUser -cConsole Main : Insert code here }
end.
```

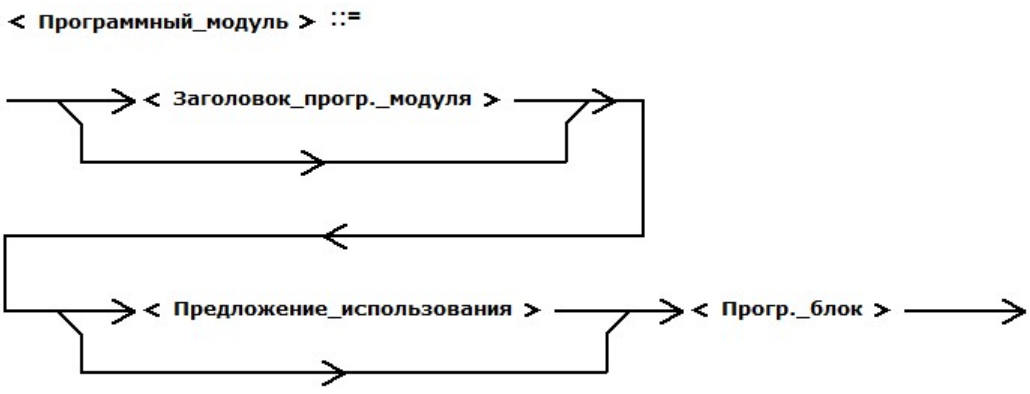
Программы, как правило, состоят из нескольких составных частей — модулей, каждая из которых отвечает за ту или иную часть логики работы программы. В Pascal/Delphi различают два *вида модулей*:

- программный модуль;
- вспомогательный модуль (модуль unit).

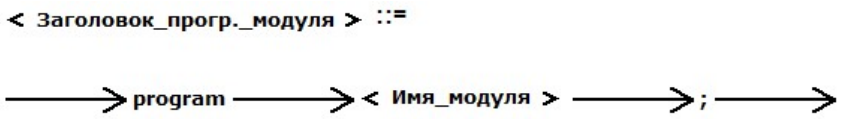
Приведённый выше текст — *программный модуль*. Каждая программа включает в себя ровно один такой модуль. В нём содержатся команды, с которых начинается выполнение программы. В зависимости от того, насколько сложна программа, она может использовать вспомогательные модули, причём как стандартные, так и разработанные самим автором программы.

В Delphi текст программного модуля сохраняется в файл с расширением .dpr или .dproj — так называемый файл проекта. Вспомогательные модули, как правило, имеют расширение .pas. Разделение на модули не только позволяет лучше отразить внутреннюю структуру программы и выделить крупные подзадачи, но и позволяет упростить командную разработку, т.к. каждый из разработчиков может заниматься разработкой своего собственного модуля независимо от других членов команды.

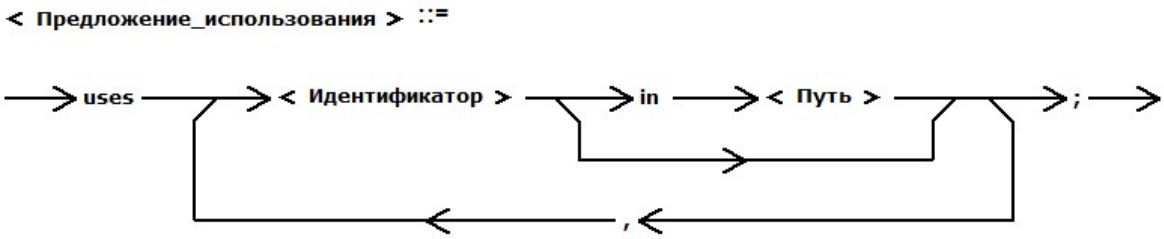
Пока остановимся только на синтаксисе программного модуля.



Формально в Pascal *заголовок программного модуля* является необязательным. Тем не менее, в Delphi его сделали обязательным, причём имя программного модуля должно совпадать с именем файла проекта, без расширения. Причина такого решения станет понятна при обсуждении вспомогательных модулей. Среда автоматически генерирует заголовок программного модуля, поэтому программисту остаётся только следить, чтобы он соответствовал требованиям и не был удалён из текста программы по ошибке.



Следующая необязательная часть программного модуля — *предложение использования uses*. Эта синтаксическая конструкция применяется в тех случаях, когда программа не ограничивается использованием только встроенных в язык средств (операций, процедур и функций), а опирается в своей работе также на вспомогательные модули.



После ключевого слова `uses` перечисляются идентификаторы (имена) модулей через запятую. Необязательная часть с ключевым словом `in` применяется только в программном модуле и позволяет задать путь к модулю явным образом: в этом случае абсолютный или относительный путь к файлу задаётся строковым литералом. Например:

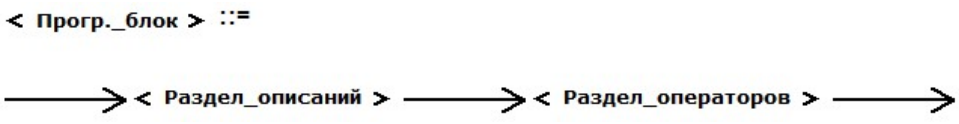
```
uses
  Windows, Messages, SysUtils, Strings in 'C:\Classes\Strings.pas',
  Classes, NFA in '..\Units\NFA.pas';
```

Вспомогательные модули, путь к которым не указан, компилятор будет искать в директории проекта, директории стандартных модулей, а также в других директориях, указанных в настройках проекта.

По умолчанию ко всем проектам подключены модули System и SysInit — их указание в предложении использования является избыточным.

Программный блок

Обязательной частью программного модуля является *программный блок*, или *тело программного модуля*.

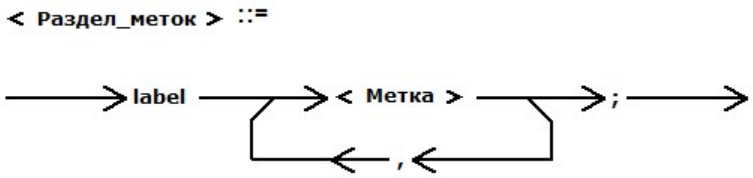


Начинается программный блок с раздела описаний (объявлений). В этой части программы размещают описания следующих элементов программы:

- меток;
- констант;
- типов;
- переменных;
- процедур и функций (подпрограмм).

В ранних версиях языка Pascal такая последовательность объявлений была обязательной, однако современные версии Pascal и все версии Delphi допускают произвольный порядок объявлений. Группы объявлений любого вида могут встречаться произвольное количество раз.

Самый простой синтаксис объявления имеют метки.



Метка — это идентификатор или целочисленный литерал, используемый для обозначения определённого места в программе и последующего к нему обращения.

При использовании в качестве меток целочисленных литералов они должны попадать в диапазон от 0 до $2^{32} - 1$.

Одно из традиционных применений меток — передача управления между частями программы (с использованием оператора goto). Это применение считается нежелательным, т.к. может усложнять чтение программы и, как следствие, затруднять поиск ошибок в ней.

Второе применение меток — задачи защиты программного обеспечения от несанкционированного копирования. Для противодействия взлому программ в этом случае могут использоваться такие приёмы, как шифрование кода, контроль целостности программ и т.п. — при реализации этих приёмов часто необходимо иметь возможность обратиться по имени к конкретным участкам кода.

Пример *раздела меток*:

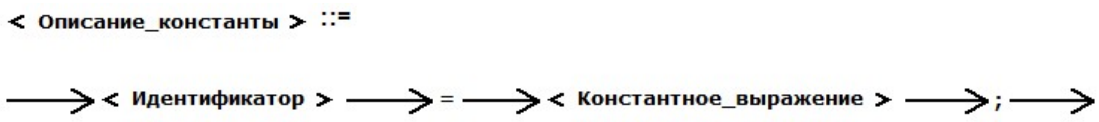
```
label blockStart, 8, blockEnd;
```

Раздел констант позволяет назначить имена используемым в программе постоянным величинам.

Из приведённой синтаксической диаграммы видно, что константы могут быть двух видов:

– типизированные;
– нетипизированные.

Описание *нетипизированной константы* имеет следующий синтаксис:



В тексте программы это выглядит, например, так:

```
const
  MaxItemCount = 200;
  DefaultUserName = 'Guest';
```

Объявление констант может быть особенно полезным, если какая-либо величина встречается в тексте программы многократно.

В приведённом выше примере объявлена константа `MaxItemCount`, которая, судя по названию, будет использоваться в программе для обозначения максимально допустимого количества каких-то элементов данных. Если вместо объявления подобной константы по всей программе использовать само значение, это создаст проблемы в будущем, когда ограничение потребуется изменить, например, на 10 000. Автоматическая замена числа 200 по всему тексту программы здесь не сработает, т.к. в некоторых местах программы число 200 может означать что-то другое — придётся внимательно просматривать текст программы и вручную выполнять замену тех чисел 200, которые действительно нужно заменить. При объявлении же константы достаточно будет во всей программе использовать её только по имени, а при изменении её значения — изменить его в единственном месте программы, в самом объявлении константы.

Для значений, используемых в программе один раз, объявление константы также может быть целесообразным. Рассмотрим такой пример:

```
(Key >= 60) and (Key <= 93)
```

Что означают числа 60 и 93? Почему именно они, а не, скажем, 48 и 115? Поможет ли уточнение, что программа, в которой записано такое выражение, — это виртуальное пианино? Что такое 60? Что такое 93?

В программировании такие величины называют *magic numbers*, или *магические числа*. Злоупотребление таким способом записи константных величин в больших проектах нередко приводит к тому, что новые сотрудники, приходя на проект, длительное время не могут приступить непосредственно к работе с ним, поскольку просто не могут понять, что означает записанный таким способом текст программы. Ещё хуже, если разработчики, которые понимали назначение этих величин, уходят из компании и не успевают передать своё знание остальным.

Сравните теперь предыдущий пример с таким:

```
(Key >= Note_C3) and (Key <= Note_A5)
```

или таким:

```
(Key >= KeyFirstVisible) and (Key <= KeyLastVisible)
```

Согласитесь, смысл начинает проясняться. В первом случае можно догадаться, что сравниваются значения, которые соответствуют каким-то нотам, а во втором — и вовсе понять, что эти значения задают номера первой и последней нот, которые можно сыграть на виртуальном пианино. Цена такой «роскоши» — 2–3 строки в разделе описаний:

```
const
  KeyFirstVisible = 60;
  KeyLastVisible = 93;
```

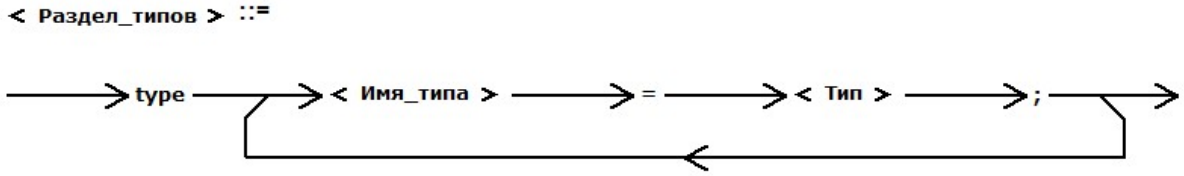
Значение константы необязательно должно быть литералом. Допускается запись так называемых *константных выражений* — выражений, значения которых могут быть вычислены без запуска программы. Как правило, это выражения, состоящие из констант, операций над ними и ограниченным набором встроенных функций.

Типизированные константы во многих реализациях языка Pascal, а также ранних версиях Delphi вели себя, как обычные переменные, но имеющие начальное значение. В более поздних версиях Delphi это поведение настраивается: можно запретить изменение типизированных констант, сделав их полноценными константами.

Синтаксис объявления типизированных констант отличается явным указанием типа, который будет иметь константа:

```
const
  MinAverageMark: Real = 4.3;
```

В приведённом примере константа будет иметь вещественный тип. Однако на практике типизированные константы чаще всего используются для объявления констант сложных типов: массивов, записей и т.п. Синтаксис объявления таких констант целесообразно рассматривать при изучении соответствующих тем.



По той же причине, по которой имеет смысл объявлять в разделе констант используемые в программе постоянные величины, нередко целесообразно объявлять и собственные типы данных в *разделе типов*.

Представьте себе, что Вы пишете программу, которая ведёт учёт товаров в магазине, торгующем пирожками с капустой. В момент разработки программы в магазине продаются только пирожки и продаются они всегда целиком, поэтому для обозначения их количества Вы решаете применить целочисленный тип данных. Например, Integer.

Спустя полгода магазин расширяет ассортимент и начинает торговать не только пирожками, но и пиццей на развес. Разумеется, если покупатель может взять себе половину пиццы, целочисленные типы здесь уже не помогут. Но как быстро внести изменения в программу, если в её тексте везде для обозначения количества товара использован Integer? Ведь в ней будут и другие данные того же типа.

А теперь предположим, что Вы не стали использовать Integer, а объявили собственный тип данных:

```
type
  TFoodAmount = Integer;
```

Смотрите: теперь, если по тексту программы для обозначения количества пирожков везде использовался тип TFoodAmount, достаточно будет изменить его объявление и немного доработать программу в тех местах, где учёт пиццы отличается от учёта пирожков. Но это будет намного меньший объём работы и намного меньшая вероятность внести ошибки в программу.

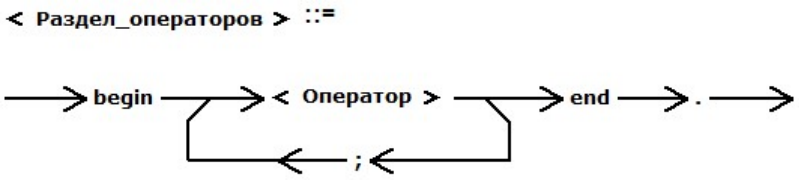
В сообществе Delphi-разработчиков существует общепринятое соглашение, что имена типов, объявленные программистом (т.е. не являющиеся встроенными), должны начинаться с буквы T (от слова type). Это необязательно, но если Вы не придерживаетесь этого стиля, Вас будут считать новичком, который ещё недостаточно опытен, чтобы знать такие тонкости.

Приведённый пример иллюстрирует синтаксис, в котором новый тип объявляется как эквивалентный другому, уже существующему типу (метапеременная «Имя_типа» на синтаксической диаграмме). Случай, когда вместо имени типа указывается задание типа, применяется к более сложным типам: массивам, записям, диапазонам, перечислимым типам и т.п. Задание этих типов целесообразно рассматривать при изучении соответствующих тем.

Раздел переменных используется для объявления переменных, которые будут использоваться в программе. Все переменные обязательно должны быть объявлены. Пример объявления:

```
var
  A, B, C: Real;
  N, M: Integer;
  Letter: Char;
  IsValidTriangle: Boolean;
```

Вторая часть программного блока — раздел операторов. Он имеет следующий синтаксис:



В *разделе операторов* записывают команды программы, которые, как можно догадаться, называются операторами. Важная синтаксическая особенность: раздел операторов завершается точкой после ключевого слова `end`.

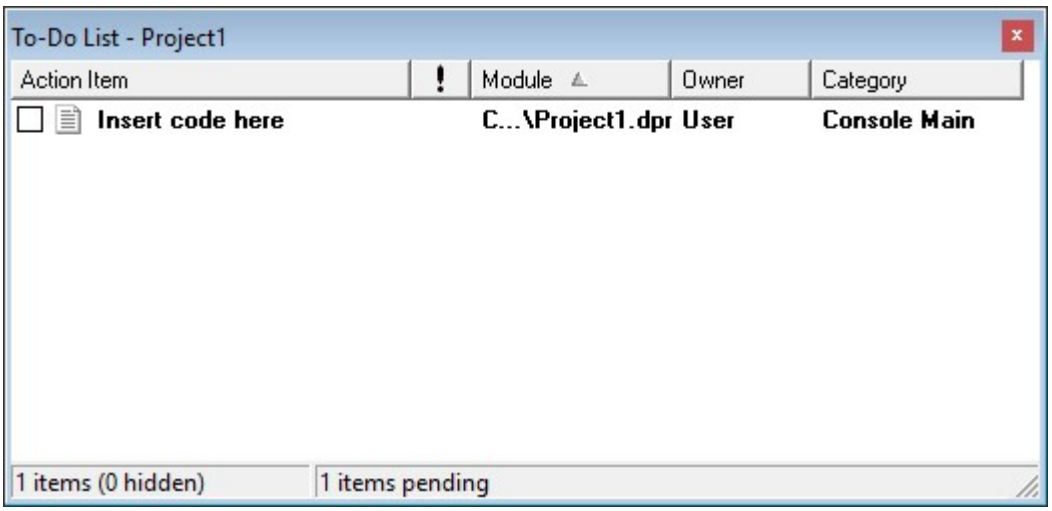
Вернёмся теперь к тексту программы, с которого начали обсуждение.

```
program Project1;

{$APPTYPE CONSOLE}

uses
  SysUtils;

begin
  { TODO -oUser -cConsole Main : Insert code here }
end.
```

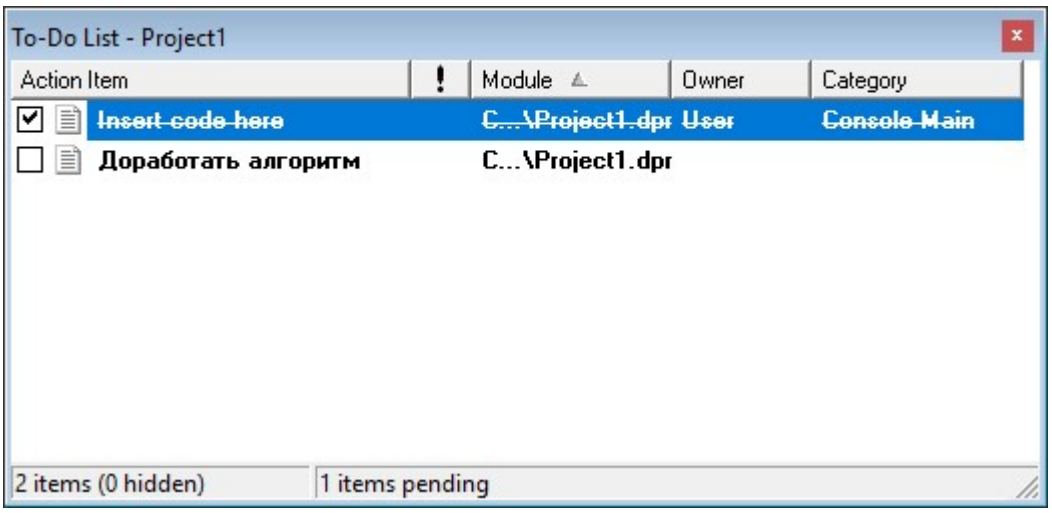
Первое, что можно в нём увидеть, — заголовок программы, который в Delphi сделали обязательным.

Также здесь присутствует предложение использования uses, в котором указан модуль SysUtils. Среда Delphi по умолчанию подключает этот модуль, потому что он предоставляет много полезных возможностей, необходимых широкому кругу программ. Тем не менее, для простых программ, обходящихся использованием только стандартных возможностей языка, этот модуль можно не подключать, а заодно и убирать предложение использования. За ненадобностью.

Также видим раздел операторов, в который вписан комментарий. Однако текст комментария выглядит немного странно. Этот комментарий составлен таким образом, чтобы его можно было увидеть в специальном окне среды программирования Delphi, которое называется To-Do List (список дел, которые необходимо сделать):



Идея заключается в том, что среда автоматически собирает по всей программе комментарии такого вида и отображает их в этом окне. Это бывает удобно, например, при использовании нисходящего проектирования, когда какая-то часть программы временно реализуется простым, но неэффективным способом или описывается, но не делает ничего полезного. В этом случае в таком месте программы можно оставить комментарий такого вида, чтобы не забыть вернуться к этому делу позже. Писать их вручную не требуется: среда программирования предоставляет инструменты для автоматической их генерации в нужных местах.



Остаётся последняя строка, смысл которой может быть непонятен:

```
{$APPTYPE CONSOLE}
```

Внешне очень похоже на комментарий, не так ли? Но почему-то среда отображает его не так, как обычные комментарии.

Это *директива компилятора* — указание компилятору о том, как следует обрабатывать программу или какую-либо её часть.

В первоначальном варианте языка Pascal директив компилятора не было. Но язык оказался настолько популярным, что вскоре появились десятки компиляторов, разработчики которых дополняли язык новыми возможностями, для которых появлялись и новые настройки компилятора. Согласитесь, удобно было бы, чтобы все настройки компиляции содержались прямо в тексте программы: тогда точно можно быть уверенным, что программа будет скомпилирована правильно.

Но была одна проблема: в самом языке не было предусмотрено возможности записывать указания компилятору. А если добавлять какую-нибудь новую синтаксическую конструкцию, её могли не понять и посчитать синтаксической ошибкой уже существовавшие компиляторы.

Как быть? Сделать директивы компилятора похожими по синтаксису на комментарии. Тогда старые компиляторы будут считать директиву обычным комментарием и проигнорируют её, а компиляторы, которые знают про директивы, смогут их распознавать и учитывать при обработке текста программы. А для того, чтобы новые компиляторы могли отличить директиву от обычного комментария, договорились, что директива будет выглядеть как комментарий, который начинается с символа `$`. Разработчикам Delphi оставалось просто позаимствовать идею директив компилятора у Borland Pascal.

В нашем примере директива `{$APPTYPE CONSOLE}` даёт компилятору указание сформировать консольное приложение — особый вид Windows-приложения, для которого операционная система автоматически создаёт текстовое окно (консоль). А ведь именно такой тип проекта мы и создавали.

Дополнительные вопросы

1. Постройте синтаксическую диаграмму для раздела описаний.

2. Опишите синтаксис приведённых конструкций языка в РБНФ.

3. В каком ещё современном языке программирования был использован приём, аналогичный директивам компилятора в Pascal?